Contents

Preface

1 Getting Started

2 Introduction

3 Displaying and Printing Images

4 Spatial Transformations

5 Image Registration

6 Neighborhood and Block Operations

7 Linear Filtering and Filter Design

8 Transforms

9 Morphological Operations

10 Analyzing and Enhancing Images

11 Region-Based Processing

14 Function Reference

Preface

This section introduces you to the Image Processing Toolbox and describes conventions used by the documentation.

What Is the Image Processing Toolbox?

The Image Processing Toolbox is a collection of functions that extend the capability of the MATLAB® numeric computing environment. The toolbox supports a wide range of image processing operations, including:

- Spatial image transformations
- Morphological operations
- Neighborhood and block operations
- Linear filtering and filter design
- Transforms
- Image analysis and enhancement
- Image registration
- Deblurring
- Region of interest operations

Many of the toolbox functions are MATLAB M-files, a series of MATLAB statements that implement specialized image processing algorithms. You can view the MATLAB code for these functions using the statement

```
type function_name
```

You can extend the capabilities of the Image Processing Toolbox by writing your own M-files, or by using the toolbox in combination with other toolboxes, such as the Signal Processing Toolbox and the Wavelet Toolbox.

For a list of the new features in Version 3.0, see the Release Notes documentation.

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with Image Processing Toolbox.

For more information about any of these products, see either

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the “products” section

The toolboxes listed below all include functions that extend MATLAB. The blocksets all include blocks that extend Simulink.

Product	Description
DSP Blockset	Design and simulate DSP systems
Mapping Toolbox	Analyze and visualize geographically based information
MATLAB	The Language of Technical Computing
Signal Processing Toolbox	Perform signal processing, analysis, and algorithm development
Wavelet Toolbox	Analyze, compress, and denoise signals and images using wavelet techniques

Configuration Notes

To determine if the Image Processing Toolbox is installed on your system, type this command at the MATLAB prompt.

```
ver
```

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

For information about installing the toolbox, see the MATLAB Installation Guide for your platform.

For the most up-to-date information about system requirements, see the system requirements page, available in the products area at The MathWorks Web site (www.mathworks.com).

About the Documentation

This section:

- Describes the structure of the Image Processing Toolbox documentation
- Credits the sources of images used in the documentation
- Explains the use of glossaries at the beginning of each major section of the documentation
- Provides pointers to other sources of information

Structure of the Documentation

The documentation is organized into these major sections:

- Chapter 1, "Getting Started" contains two step-by-step examples that will help you get started with using the Image Processing Toolbox.
- Chapter 2, "Introduction" introduces the Image Processing Toolbox and its capabilities.
- Chapter 3, "Displaying and Printing Images" describes how to display and print images in MATLAB.
- Chapter 4, "Spatial Transformations" describes image cropping, resizing, rotating, and other geometric transformations you can perform with the Image Processing Toolbox.
- Chapter 5, "Image Registration" describes how to align two images of the same scene using the Control Point Selection Tool.
- Chapter 6, "Neighborhood and Block Operations" describes how to perform block operations on images.
- Chapter 7, "Linear Filtering and Filter Design" describes how to create filters.
- Chapter 8, "Transforms" discusses several important image transforms.
- Chapter 9, "Morphological Operations" describes the functions in the toolbox that you can use to implement morphological image processing operations.
- Chapter 10, "Analyzing and Enhancing Images" discusses working with image data and displaying images in MATLAB and the Image Processing Toolbox.

- Chapter 11, “Region-Based Processing” describes how to perform image processing on specific regions of an image.
- Chapter 12, “Image Deblurring” describes the toolbox deblurring functions.
- Chapter 13, “Color” describes how to handle color images.

For detailed reference descriptions of each toolbox function, go to the MATLAB Help browser. Many reference descriptions also include examples, a description of the function’s algorithm, and references to additional reading material.

Image Credits

This table lists the copyright owners of the images used in the Image Processing Toolbox documentation.

Image	Source
`cameraman`	Copyright Massachusetts Institute of Technology. Used with permission.
`cell`	Cancer cell from a rat’s prostate, courtesy of Alan W. Partin, M.D., Ph.D., Johns Hopkins University School of Medicine.
`circuit`	Micrograph of 16-bit A/D converter circuit, courtesy of Steve Decker and Shujaat Nadeem, MIT, 1993.
`concordaerial` and `westconcordaerial`	Visible color aerial photographs courtesy of mPower3/Emerge.
`concordorthophoto` and `westconcordorthophoto`	Orthoregistered photographs courtesy of Massachusetts Executive Office of Environmental Affairs, MassGIS.
`forest`	Photograph of Carmanah Ancient Forest, British Columbia, Canada, courtesy of Susan Cohen.

Image	Source
LAN files	Permission to use Landsat™ data sets provided by Space Imaging, LLC, Denver, Colorado.
m83	M83 spiral galaxy astronomical image courtesy of Anglo-Australian Observatory, photography by David Malin.
moon	Copyright Michael Myers. Used with permission.
trees	*Trees with a View,* watercolor and ink on paper, copyright Susan Cohen. Used with permission.

In addition, the following images are copyrighted:

J. C. Russ, *The Image Processing Handbook,* Third Edition, 1998, CRC Press, Boca Raton, ISBN 0-8493-2532-3. Used with permission.

afmsurf	enamel	rice
alumgrns	flowers	saturn
bacteria	ic	shot1
blood1	lily	testpat1
bonemarr	ngc4024l	testpat2
circles	ngc4024m	text
circlesm	ngc4024s	tire
debye1	pearlite	tissue1

Terminology

At the beginning of each chapter (and sometimes at the beginning of a major section within a chapter) are tables that serve as glossaries of words you need to know to understand the information in the chapter. These tables clarify how

we use terms that may be used in several different ways in image processing literature. For example:

- Sometimes in the field of image processing, one word is used to describe more than one concept. For example the *resolution* of an image can describe the height and width of an image as a quantity of pixels in each direction, or it can describe the number of pixels per linear measure, such as 100 dots per inch.
- Sometimes in the field of image processing, the same concepts are described by different terminology. For example, a *grayscale* image can also be called an *intensity* image.

MATLAB Newsgroup

If you read newsgroups on the Internet, you might be interested in the MATLAB newsgroup (`comp.soft-sys.matlab`). This newsgroup gives you access to an active MATLAB user community. It is an excellent way to seek advice and to share algorithms, sample code, and M-files with other MATLAB users.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, user input, items in drop-down lists	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

Image Processing Demos

The Image Processing Toolbox is supported by a full complement of demo applications. These are very useful as templates for your own end-user applications, or for seeing how to use and combine your toolbox functions for powerful image analysis and enhancement. The toolbox demos are located under the subdirectory,

```
matlabroot\toolbox\images\imdemos
```

where `matlabroot` represents your MATLAB installation directory.

The table below lists the demos available.

The easiest way to run an individual demo is to enter its name at the MATLAB command prompt. You can also launch MATLAB demos from the MATLAB Demo Window. To evoke this window select **Demos** from the **Help** menu of the main MATLAB window, or simply type `demo` at the command prompt. To see the list of available image processing demos, double-click on **Toolboxes** from the list on the left, then select **Image Processing**. Select the desired demo and press the **Run** button.

If you want to know whether there is a demo that uses a particular function, check the function name in the index. If there is a demo that demonstrates this function, a subentry of "See also `demoname`" will appear, where `demoname` is the name of the demo.

Demo Name	Brief Description
`dctdemo`	DCT image compression: you choose the number of coefficients and it shows you a reconstructed image and an error image.
`edgedemo`	Edge detection: all supported types with optional manual control over threshold, direction, and sigma, as appropriate to the method used.
`firdemo`	2-D Finite Impulse Response (FIR) filters: design your own filter by changing the cut-off frequency and filter order.

`imadjdemo`	Contrast Adjustment and Histogram Equalization: adjust intensity values using brightness, contrast, and gamma correction, or by using histogram equalization.
`ipexconformal`	Explore a Conformal Mapping: illustrates how to use spatial- and image-transformation functions to perform a conformal mapping.
`ipexdeconvblind`	Deblurring Images Using the Blind Deconvolution Algorithm: illustrates use of the `deconvblind` function.
`ipexdeconvlucy`	Deblurring Images Using the Lucy-Richardson Algorithm: illustrates use of the `deconvlucy` function.
`ipexdeconvreg`	Deblurring Images Using a Regularized Filter: illustrates use of the `deconvreg` function.
`ipexdeconvwnr`	Deblurring Images Using the Wiener Filter: illustrates use of the `deconvwnr` function.
`ipexgranulometry`	Finding the Granulometry of Stars in an Image: illustrates how to use morphology functions to perform granulometry.
`ipexmri`	Extracting Slices from a 3-Dimensional MRI Data Set: illustrates how to use the image transformation functions to interpolate and reslice a three-dimensional MRI data set, providing a convenient way to view a volume of data.
`ipexnormxcorr2`	Registering an Image Using Normalized Cross-correlation: illustrates how to use translation to align two images.
`ipexregaerial`	Registering an Aerial Photo to an Orthophoto: illustrates how to use the Control Point Selection Tool to align two images.

`ipexrotate`	Finding the Rotation and Scale of a Distorted Image: illustrates how to use the `cp2tform` function to get the rotation angle and scale factor of a distorted image.
`ipexsegcell`	Detecting a Cell Using Image Segmentation: illustrates how to use dilation and erosion to perform edge detection.
`ipexsegmicro`	Detecting Microstructures Using Image Segmentation: illustrates how to use morphological opening and closing to extract large objects from an image.
`ipexsegwatershed`	Detecting Touching Objects Using Watershed Segmentation: illustrates use of morphology functions to perform marker-control watershed segmentation.
`ipexshear`	Padding and Shearing an Image Simultaneously: illustrates how to use the padding options of the image transformation functions.
`ipextform`	Creating a Gallery of Transformed Images: illustrates how to use the `imtransform` function to perform many types of image transformations.
`ipss001`	Connected Components Labelling: includes double thresholding, feature-based logic, and binary morphology. All operations are performed on one image.
`ipss002`	Feature-based Logic: Shows object selection using `AND` operations on the 'on' pixels in two binary images and shows filtering and thresholding on a single image.
`ipss003`	Correction of Nonuniform Illumination: creates a coarse approximation of the background, subtracts it from the image, and then adjusts the pixel intensity values to fill the entire range.

`nrfiltdemo`	Noise Reduction Using Linear and Non-linear Filters: allows you to add different types of noise with variable densities, and choose a filter neighborhood size.
`qtdemo`	Quadtree Decomposition: select a threshold and see a representation of the sparse matrix and a reconstruction of the original image.
`roidemo`	Region of Interest (ROI) Selection: select an ROI and apply operations such as unsharp and fill. Also displays the binary mask of the ROI.

1

Getting Started

This section contains two examples to get you started doing image processing using MATLAB and the Image Processing Toolbox. The examples contain cross-references to other sections in this manual that have in-depth discussions on the concepts presented in the examples.

Example 1 — Some Basic Topics

Before beginning with this exercise, start MATLAB. If you are new to MATLAB, you should first read the MATLAB Getting Started documentation.

You should already have installed the Image Processing Toolbox, which runs seamlessly from MATLAB. For information about installing the toolbox, see the MATLAB Installation Guide for your platform.

All of the images used in this example are supplied with the Image Processing Toolbox. Note that the images shown in this documentation differ slightly from what you see on your screen because the surrounding MATLAB figure window has been removed to save space.

1. Read and Display an Image

Clear the MATLAB workspace of any variables and close open figure windows.

```
clear, close all
```

To read an image, use the `imread` command. Let's read in a TIFF image named `pout.tif` (which is one of the sample images that is supplied with the Image Processing Toolbox), and store it in an array named `I`.

```
I=imread('pout.tif');
```

Now call `imshow` to display `I`.

```
imshow(I)
```



Here's What Just Happened

Step 1. The `imread` function recognized `pout.tif` as a valid TIFF file and stored it in the variable `I`. (For the list of graphics formats supported, see `imread` in the Image Processing Toolbox online "Function Reference.")

The functions `imread` and `imshow` read and display graphics images in MATLAB. In general, it is preferable to use `imshow` for displaying images because it handles the image-related MATLAB properties for you. (The MATLAB function `image` is for low-level programming tasks.)

Note that if `pout.tif` were an *indexed* image, the appropriate syntax for `imread` would be,

```
[X, map] = imread('pout.tif');
```

(For more information on the supported image types, see "Image Types in the Toolbox" on page 2-5.)

2. Check the Image in Memory

Enter the `whos` command to see how `I` is stored in memory.

```
whos
```

MATLAB responds with

```
Name      Size         Bytes  Class

  I       291x240        69840  uint8 array

Grand total is 69840 elements using 69840 bytes
```

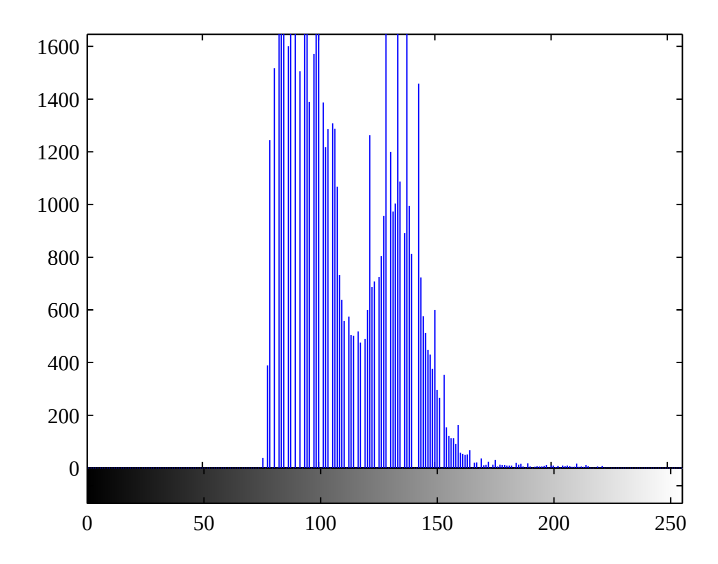
Here's What Just Happened

Step 2. You called the `whos` command to see how `pout.tif` had been stored into the MATLAB workspace. As you saw, `pout.tif` is stored as a 291-by-240 array. Since `pout.tif` was an 8-bit image, it gets stored in memory as an `uint8` array. MATLAB can store images in memory as `uint8`, `uint16`, or `double` arrays. (See "Reading a Graphics Image" on page 2-16 for an explanation of when the different storage classes are used.)

3. Perform Histogram Equalization

As you can see, `pout.tif` is a somewhat low contrast image. To see the distribution of intensities in `pout.tif` in its current state, you can create a histogram by calling the `imhist` function. (Precede the call to `imhist` with the `figure` command so that the histogram does not overwrite the display of the image `I` in the current figure window.)

```
figure, imhist(I) % Display a histogram of I in a new figure.
```



Notice how the intensity range is rather narrow. It does not cover the potential range of [0, 255], and is missing the high and low values that would result in good contrast.

Now call `histeq` to spread the intensity values over the full range, thereby improving the contrast of `I`. Return the modified image in the variable `I2`.

```
I2 = histeq(I); % Equalize I and output in new array I2.
```

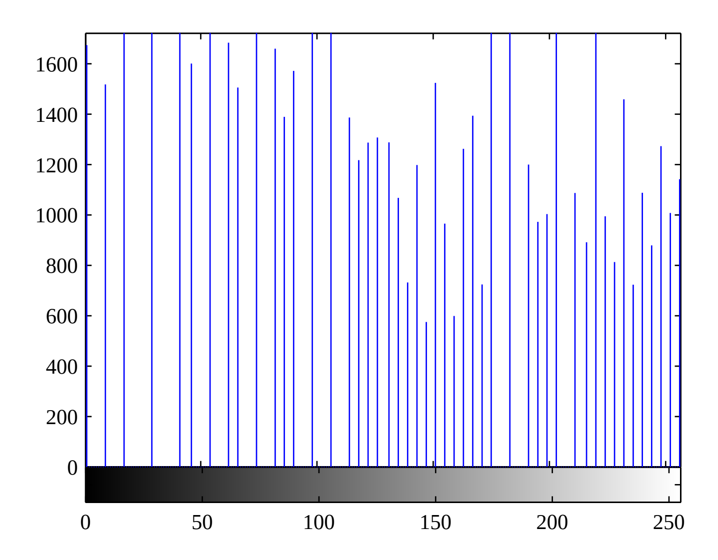
Display the new equalized image, `I2`, in a new figure window.

```
figure, imshow(I2)  % Display the new equalized image I2.
```



Call `imhist` again, this time for `I2`.

```
figure, imhist(I2) % Show the histogram for the new image I2.
```



See how the pixel values now extend across the full range of possible values.

Here's What Just Happened

Step 3. You adjusted the contrast automatically by using the function `histeq` to evenly distribute the image's pixel values over the full potential range for the storage class of the image. For an image `X`, with a storage class of `uint8`, the full range is $0 \le X \le 255$, for `uint16` it is $0 \le X \le 65535$, and for `double` it is $0 \le X \le 1$. Note that the convention elsewhere in this user guide (and for all MATLAB documentation) is to denote the above ranges as [0,255], [0,65535], and [0,1], respectively.

If you compare the two histograms, you can see that the histogram of `I2` is more spread out and flat than the histogram of `I1`. The process that flattened and spread out this histogram is called *histogram equalization*.

For more control over adjusting the contrast of an image (for example, if you want to choose the range over which the new pixel values should span), you can use the `imadjust` function, which is demonstrated under "5. Adjust the Image Contrast" on page 1-13 in Exercise 2.

4. Write the Image

Write the newly adjusted image `I2` back to disk. Let's say you'd like to save it as a PNG file. Use `imwrite` and specify a filename that includes the extension `'png'`.

```
imwrite (I2, 'pout2.png');
```

Here's What Just Happened

Step 4. MATLAB recognized the file extension of `'png'` as valid and wrote the image to disk. It wrote it as an 8-bit image by default because it was stored as a `uint8` intensity image in memory. If `I2` had been an image array of type RGB and class `uint8`, it would have been written to disk as a 24-bit image. If you want to set the bit depth of your output image, use the `BitDepth` parameter with `imwrite`. This example writes a 4-bit PNG file.

```
imwrite(I2, 'pout2.png', 'BitDepth', 4);
```

Note that all output formats do not support the same set of output bit depths. See `imwrite` in the Reference for the list of valid bit depths for each format. See also "Writing a Graphics Image" on page 2-17 for a tutorial discussion on writing images using the Image Processing Toolbox.

5. Check the Contents of the Newly Written File

Now, use the `imfinfo` function to see what was written to disk. Be sure not to end the line with a semicolon so that MATLAB displays the results. Also, be sure to use the same path (if any) as you did for the call to `imwrite`, above.

```
imfinfo('pout2.png')
```

MATLAB responds with

```
ans =
           Filename:'pout2.png'
        FileModDate:'03-Jun-1999 15:50:25'
           FileSize:36938
             Format:'png'
      FormatVersion:[]
```

```
       Width:240
      Height:291
    BitDepth:8
   ColorType:'grayscale'
   . . .
```

Note This example shows only a subset of all the fields returned by `imfinfo`. Also, the value in the `FileModDate` field for your file will be different from what is shown above. It will show the date and time that you used `imwrite` to create your image.

Here's What Just Happened
Step 5. When you called `imfinfo`, MATLAB displayed all of the header fields for the PNG file format that are supported by the toolbox. You can modify many of these fields by using additional parameters in your call to `imwrite`. The additional parameters that are available for each file format are listed in tables in the reference entry for `imwrite`. (See "Querying a Graphics File" on page 2-18 for more information about using `imfinfo`.)

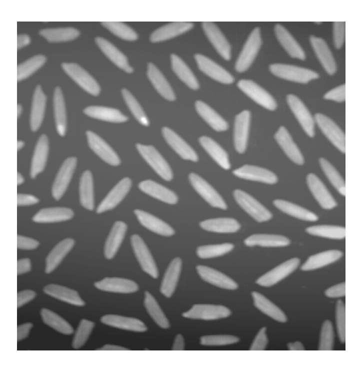
Example 2 — Advanced Topics

In this exercise you will work with another intensity image, `rice.tif`, and explore some more advanced operations. The goals of this exercise are to remove the nonuniform background from `rice.tif`, convert the resulting image to a binary image by using thresholding, use components labeling to return the number of objects (grains or partial grains) in the image, and compute object statistics.

1. Read and Display an Image

Clear the MATLAB workspace of any variables and close open figure windows. Read and display the intensity image `rice.tif`.

```
clear, close all
I = imread('rice.tif');
imshow(I)
```



2. Use Morphological Opening to Estimate the Background

Notice that the background illumination is brighter in the center of the image than at the bottom. Use the `imopen` function to estimate the background illumination.

```
background = imopen(I,strel('disk',15));
```

To see the estimated background image, type

```
imshow(background)
```

Here's What Just Happened

Step 1. You used the toolbox functions `imread` and `imshow` to read and display an 8-bit intensity image. `imread` and `imshow` are discussed in Exercise 1, in "2. Check the Image in Memory" on page 1-3, under the "Here's What Just Happened" discussion.

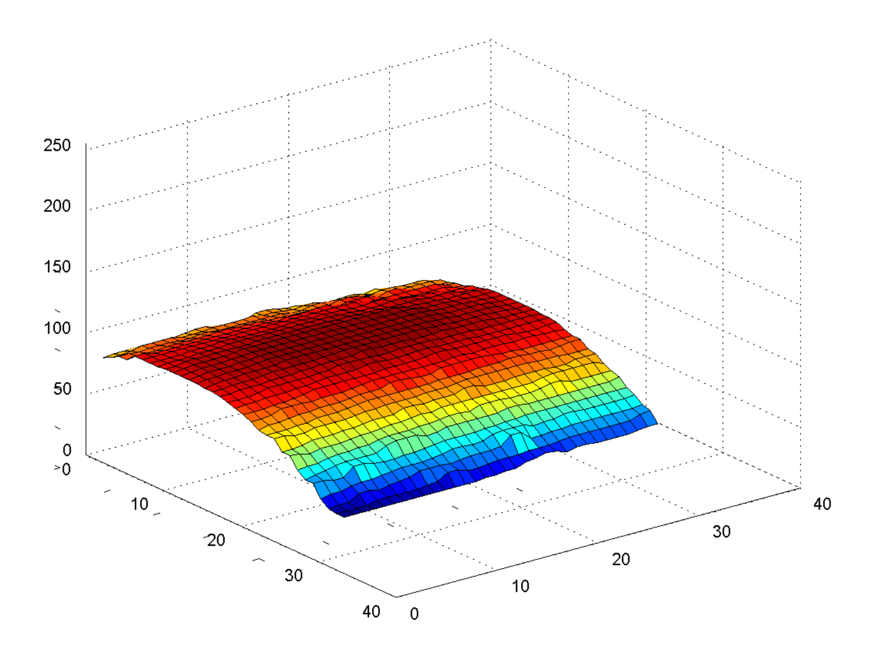
Step 2. You performed a morphological opening operation by calling imopen with the input image, `I`, and a disk-shaped structuring element with a radius of 15. The structuring element was created by the `strel` function. The morphological opening has the effect of removing objects that cannot completely contain a disk of radius 15. For more information about morphological opening, see Chapter 9, "Dilation- and Erosion-Based Functions."

3. Display the Background Approximation as a Surface

Use the `surf` command to create a surface display of the background approximation, `background`. The `surf` function requires data of class `double`, however, so you first need to convert `background` using the `double` command.

```
figure, surf(double(background(1:8:end,1:8:end))),zlim([0 255]);
set(gca,'ydir','reverse');
```

The example uses MATLAB indexing syntax to view only 1 out of 8 pixels in each direction; otherwise the surface plot would be too dense. The example also sets the scale of the plot to better match the range of the `uint8` data and reverses the y-axis of the display to provide a better view of the data (the pixels at the bottom of the image appear at the front of the surface plot).



Here's What Just Happened
Step 3. You used the `surf` command to examine the background image. The `surf` command creates colored parametric surfaces that enable you to view mathematical functions over a rectangular region. In the surface display, [0, 0] represents the origin, or upper-left corner of the image. The highest part of the curve indicates that the highest pixel values of `background` (and consequently `rice.tif`) occur near the middle rows of the image. The lowest pixel values occur at the bottom of the image and are represented in the surface plot by the lowest part of the curve. The surface plot is a Handle Graphics® object, and you can therefore fine-tune its appearance by setting properties. For information on working with MATLAB graphics, see the MATLAB graphics documentation.

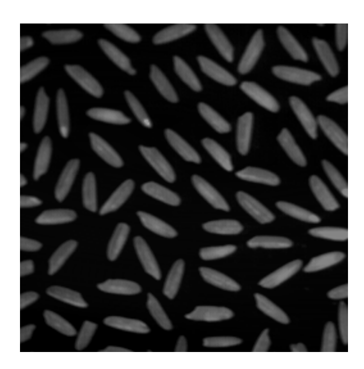
4. Subtract the Background Image from the Original Image

Now subtract the background image, `background`, from the original image, `I`, to create a more uniform background.

```
I2 = imsubtract(I,background);
```

Now display the image with its more uniform background.

```
figure, imshow(I2)
```



Here's What Just Happened
Step 4. You subtracted a background approximation image from `rice.tif`. Because subtraction, like many of MATLAB mathematical operations, is only supported for data of class `double`, you must use the Image Processing Toolbox image arithmetic `imsubtract` function. The Image Processing Toolbox has a demo, `ipss003`, that approximates and removes the background from an image. For information on how to run this (and other demos), see "Image Processing Demos" in the Preface.

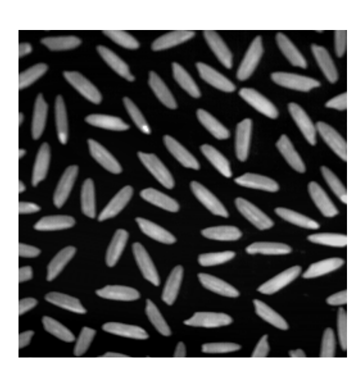
5. Adjust the Image Contrast

The image is now a bit too dark. Use `imadjust` to adjust the contrast.

```
I3 = imadjust(I2, stretchlim(I2), [0 1]);
```

Display the newly adjusted image.

```
figure, imshow(I3);
```

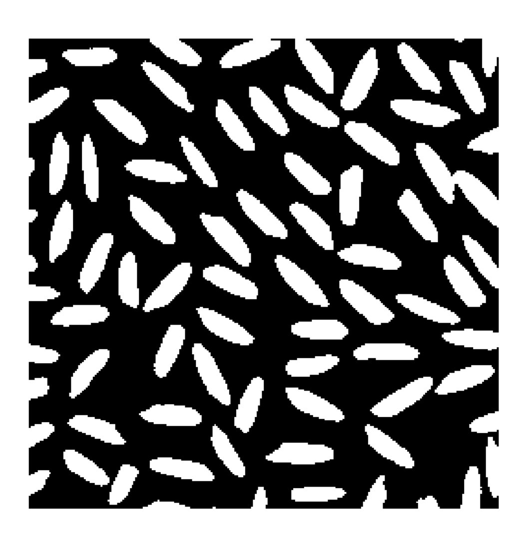


Here's What Just Happened
Step 5. You used the `imadjust` command to increase the contrast in the image. The `imadjust` function takes an input image and can also take two vectors: `[low high]` and `[bottom top]`. The output image is created by mapping the value `low` in the input image to the value `bottom` in the output image, mapping the value `high` in the input image to the value `top` in the output image, and linearly scaling the values in between. See the reference pages for `imadjust` for more information. You called `imadjust` with `stretchlim(I2)` as the second argument. The `stretchlim` function automatically computes the right `[low high]` values to make `imadjust` increase (stretch) the contrast of the image.

6. Apply Thresholding to the Image

Create a new binary thresholded image, bw, by using the functions graythresh and im2bw.

```
level = graythresh(I3);
bw = im2bw(I3,level);
figure, imshow(bw)
```



Now call the whos command to see what type of array the thresholded image bw is.

```
whos
```

MATLAB responds with

```
Name                    Size          Bytes  Class

I                    256x256          65536  uint8 array
I2                   256x256          65536  uint8 array
I3                   256x256          65536  uint8 array
background           256x256          65536  uint8 array
bw                   256x256          65536  logical array
level                  1x1                8  double array

Grand total is 327681 elements using 327688 bytes
```

Here's What Just Happened

Step 6. You called `graythresh` to automatically compute an appropriate threshold to use to convert the intensity image to binary. You then called `im2bw` to perform for thresholding, using the threshold, `level`, returned by graythresh.

Notice that when you call the `whos` command, you see the expression `logical` listed after the class for `bw`. This indicates the presence of a logical flag. The flag indicates that `bw` is a logical matrix, and the Image Processing Toolbox treats logical matrices as binary images. Thresholding using MATLAB logical operators always results in a logical image. For more information about binary images and the logical flag, see "Binary Images" on page 2-8.

7. Determine the Number of Objects in the Image

To determine the number of grains of rice in the image, use the `bwlabel` function. This function labels all of the connected components in the binary image `bw` and returns the number of objects it finds in the image in the output value, `numobjects`.

```
[labeled,numObjects] = bwlabel(bw,4);% Label components.

numObjects =

     80
```

The accuracy of your results depends on a number of factors, including:

- The size of the objects
- The accuracy of your approximated background
- Whether you set the connectivity parameter to 4 or 8
- Whether or not any objects are touching (in which case they may be labeled as one object) In the example, some grains of rice are touching, so `bwlabel` treats them as one object.

Here's What Just Happened

Step 7. You called bwlabel to search for connected components and label them with unique numbers. bwlabel takes a binary input image and a value specifying the *connectivity* of objects. The parameter 4, passed to the bwlabel function, means that pixels must touch along an edge to be considered connected. For more information about the connectivity of objects, see "Pixel Connectivity" in Chapter 9.

You can also determine the number of objects in a label matrix by asking for the maximum pixel value in the image. For example,

```
max(labeled(:))

ans =

  80
```

8. Examine the Label Matrix

You may find it helpful to take a closer look at labeled to see what bwlabel has created. Use the imcrop command to select and display pixels in a region of labeled that includes an object and some background.

To ensure that the output is displayed in the MATLAB window, do not end the line with a semicolon. In addition, choose a small rectangle for this exercise, so that the displayed pixel values don't wrap in the MATLAB command window.

The syntax shown below makes imcrop work interactively. Your mouse cursor becomes a cross-hair when placed over the image. Click at a position in labeled where you would like to select the upper left corner of a region. Drag the mouse to create the selection rectangle, and release the button when you are done.

```
grain = imcrop(labeled) % Crop a portion of labeled.
```

We chose the left edge of a grain and got the following results.

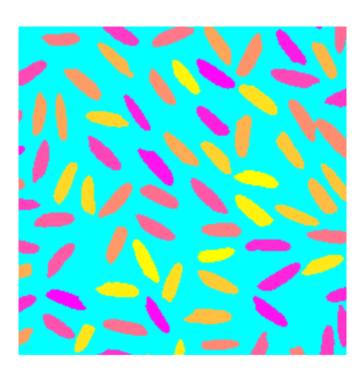
```
grain =
```

```
     0     0     0     0     0     0     0    60    60
     0     0     0     0     0    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0    60    60    60    60    60    60
     0     0     0     0     0    60    60    60    60
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
```

A good way to view a label matrix is to display it as a pseudo-color indexed image. In the pseudo-color image, the number that identifies each object in the label matrix maps to a different color in the associated colormap matrix. When you view a label matrix as an RGB image, the objects in the image are easier to distinguish.

To view a label matrix in this way, use the `label2rgb` function. Using this function, you can specify the colormap, the background color, and how objects in the label matrix map to colors in the colormap.

```
RGB_label = label2rgb(labeled, @spring, 'c', 'shuffle');
imshow(RGB_label);
```



Here's What Just Happened

Step 8. You called `imcrop` and selected a portion of the image that contained both some background and part of an object. The pixel values were returned in the MATLAB window. If you examine the results above, you can see the corner of an object labeled with 60's, which means that it was the 60th object labeled by `bwlabel`.

The `imcrop` function can also take a vector specifying the coordinates for the crop rectangle. In this case, it does not operate interactively. For example, this call specifies a crop rectangle whose upper-left corner begins at (15, 25) and has a height and width of 10.

```
rect = [15 25 10 10];
roi = imcrop(labeled, rect)
```

You are not restricted to rectangular regions of interest. The toolbox also has a `roipoly` command that enables you to select polygonal regions of interest. Many image processing operations can be performed on regions of interest, including filtering and filling. See Chapter 11, "Region-Based Processing" for more information.

The call to `label2rgb` illustrates a good way to visualize label matrices. The pixel values in the label matrix are used as indices into a colormap. Using `label2rgb`, you can specify your own colormap or use one of the MATLAB colormap-creating functions, including `gray`, `pink`, `spring`, and `hsv`. For information on these functions, see `colormap` in the MATLAB *Function Reference*.

9. Measure Object Properties in the Image

The `regionprops` command measures object or region properties in an image and returns them in a structure array. When applied to an image with labeled components, it creates one structure element for each component. Use `regionprops` to create a structure array containing some basic properties for `labeled`.

```
graindata = regionprops(labeled,'basic')
```

MATLAB responds with

```
graindata =

80x1 struct array with fields:
    Area
    Centroid
    BoundingBox
```

To find the area of the component labeled with 51's, use dot notation to access the `Area` field in the 51st element in the `graindata` structure array. Note that structure field names are case sensitive, so you need to capitalize the name as shown.

```
graindata(51).Area
```

returns the following results

```
ans =

   296
```

To find the smallest possible bounding box and the centroid (center of mass) for the same component, use this code:

```
graindata(51).BoundingBox, graindata(51).Centroid

ans =

  142.5000   89.5000   24.0000   26.0000
ans =

  155.3953  102.1791
```

To create a new vector, `allgrains`, which holds just the area measurement for each grain, use this code:

```
allgrains = [graindata.Area];
whos allgrains
```

Call the `whos` command to see how MATLAB allocated the `allgrains` variable.

```
  Name               Size          Bytes  Class

  allgrains          1x80            640  double array
```

```
Grand total is 80 elements using 640 bytes
```

allgrains is a one-row array of 80 elements, where each element contains the area measurement of a grain. Check the area of the 51st element of allgrains.

```
allgrains(51)
```

returns

```
ans =

   296
```

which is the same result that you received when using dot notation to access the Area field of graindata(51).

Here's What Just Happened
Step 9. You called regionprops to return a structure of basic property measurements for each thresholded grain of rice. The regionprops function supports many different property measurements, but setting the properties parameter to 'basic' is a convenient way to return three of the most commonly used measurements: the area, the centroid (or center of mass), and the bounding box. The bounding box represents the smallest rectangle that can contain a region, or in this case, a grain. The four-element vector returned by the BoundingBox field, `[142.5000   89.5000   24.0000   26.0000]` shows that the upper left corner of the bounding box is positioned at [142.5 89.5], and the box has a width of 24.0 and a height of 26.0. (The position is defined in spatial coordinates, hence the decimal values. For more information on the spatial coordinate system, see "Spatial Coordinates" on page 2-29.) For more information about working with MATLAB structure arrays, see "Structures" in the MATLAB programming and data types documentation.
You used dot notation to access the Area field of all of the elements of graindata and stored this data to a new vector allgrains. This step simplifies analysis made on area measurements because you do not have to use field names to access the area.

10. Compute Statistical Properties of Objects in the Image

Now use MATLAB functions to calculate some statistical properties of the thresholded objects. First use `max` to find the size of the largest grain. (If you have followed all of the steps in this exercise, the "largest grain" is actually two grains that are touching and have been labeled as one object).

```
max(allgrains)
```

returns

```
ans =

   695
```

Use the `find` command to return the component label of this large-sized grain.

```
biggrain = find(allgrains==695)
```

returns

```
biggrain =

    68
```

Find the mean grain size.

```
mean(allgrains)
```

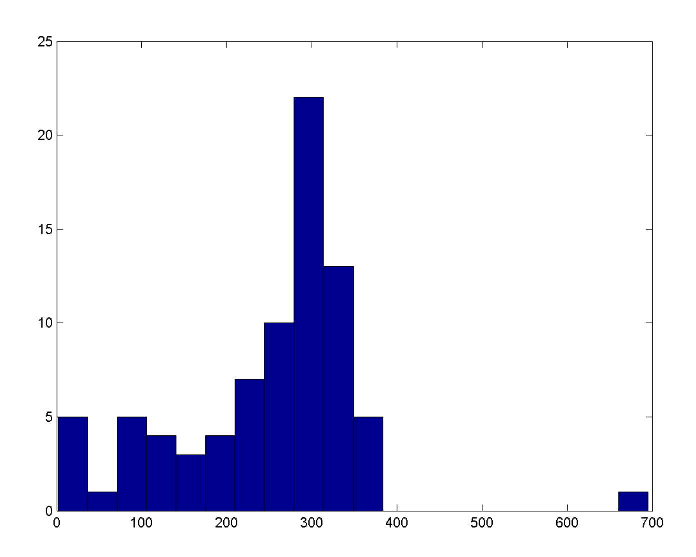
returns

```
ans =

   249
```

Make a histogram containing 20 bins that show the distribution of rice grain sizes.

```
hist(allgrains,20)
```



Here's What Just Happened

Step 10. You used some of the MATLAB statistical functions, `max`, `mean`, and `hist` to return the statistical properties for the thresholded objects in `rice.tif`.

The Image Processing Toolbox also has some statistical functions, such as `mean2` and `std2`, which are well suited to image data because they return a single value for two-dimensional data. The functions `mean` and `std` were suitable here because the data in `allgrains` was one dimensional.

The histogram shows that the most common sizes for rice grains in this image are in the range of 300 to 400 pixels.

Where to Go from Here

For more information about the topics covered in these exercises, read the tutorial chapters that make up the remainder of this documentation. For reference information about any of the Image Processing Toolbox functions, see the online "Function Reference", which complements the M-file help that is displayed in the MATLAB command window when you type

```
help functionname
```

For example,

```
help imshow
```

Online Help

The *Image Processing Toolbox User's Guide* is available online in both HTML and PDF formats. To access the HTML help, select **Help** from the menu bar of the MATLAB desktop. In the Help browser, expand the **Image Processing Toolbox** topic in the list. To access the PDF help, click on **Image Processing Toolbox** in the **Contents** tab of the Help browser, and go to the link under "Printable Documentation (PDF)." (Note that to view the PDF help, you must have Adobe's Acrobat Reader installed.)

Toolbox Demos

The Image Processing Toolbox includes many demo applications. The demos are useful for seeing the toolbox features put into action and for borrowing code for your own applications. See "Image Processing Demos" in the Preface for a complete list and summary of the demos, as well as instructions on how to run them.

2

Introduction

This section introduces you to the fundamentals of image processing using MATLAB and the Image Processing Toolbox. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Binary image	An image containing only black and white pixels. In MATLAB, a binary image is represented as a logical array of 0's and 1's (which usually represent black and white, respectively). This documentation often uses the variable name `BW` to represent a binary image in memory.
Image type	The defined relationship between array values and pixel colors. The toolbox supports binary, indexed, intensity, and RGB image types.
Indexed image	An image whose pixel values are direct indices into an RGB colormap. In MATLAB, an indexed image is represented by an array of class `uint8`, `uint16`, or `double`. The colormap is always an m-by-3 array of class `double`. We often use the variable name `X` to represent an indexed image in memory, and `map` to represent the colormap.
Intensity image	An image consisting of intensity (grayscale) values. In MATLAB, intensity images are represented by an array of class `uint8`, `uint16`, or `double`. While intensity images are not stored with colormaps, MATLAB uses a system colormap to display them. We often use the variable name `I` to represent an intensity image in memory. This term is synonymous with the term *grayscale*.
Multiframe image	An image file that contains more than one image, or *frame*. When in MATLAB memory, a multiframe image is a 4-D array where the fourth dimension specifies the frame number. This term is synonymous with the term *multipage* image.

Terms	Definitions
RGB image	An image in which each pixel is specified by three values—one each for the red, green, and blue components of the pixel's color. In MATLAB, an RGB image is represented by an m-by-n-by-3 array of class `uint8`, `uint16`, or `double`. We often use the variable name `RGB` to represent an RGB image in memory.
Storage class	The numeric storage class used to store an image in MATLAB. The storage classes used in MATLAB are `uint8`, `uint16`, and `double`. Some function descriptions in the reference chapter of this User's Guide have a section entitled "Class Support" that specifies which image classes the function can operate on. When this section is absent, the function can operate on all supported storage classes.

Images in MATLAB and the Image Processing Toolbox

The basic data structure in MATLAB is the *array*, an ordered set of real or complex elements. This object is naturally suited to the representation of *images*, real-valued ordered sets of color or intensity data.

MATLAB stores most images as two-dimensional arrays (i.e., matrices), in which each element of the matrix corresponds to a single *pixel* in the displayed image. (Pixel is derived from *picture element* and usually denotes a single dot on a computer display.) For example, an image composed of 200 rows and 300 columns of different colored dots would be stored in MATLAB as a 200-by-300 matrix. Some images, such as RGB, require a three-dimensional array, where the first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

This convention makes working with images in MATLAB similar to working with any other type of matrix data, and makes the full power of MATLAB available for image processing applications. For example, you can select a single pixel from an image matrix using normal matrix subscripting.

```
I(2,15)
```

This command returns the value of the pixel at row 2, column 15 of the image `I`.

Storage Classes in the Toolbox

By default, MATLAB stores most data in arrays of class `double`. The data in these arrays is stored as double precision (64-bit) floating-point numbers. All MATLAB functions work with these arrays.

For image processing, however, this data representation is not always ideal. The number of pixels in an image may be very large; for example, a 1000-by-1000 image has a million pixels. Since each pixel is represented by at least one array element, this image would require about 8 megabytes of memory.

To reduce memory requirements, MATLAB supports storing image data in arrays as 8-bit or 16-bit unsigned integers, class `uint8` and `uint16`. These arrays require one eighth or one fourth as much memory as `double` arrays.

Image Types in the Toolbox

The Image Processing Toolbox supports four basic types of images:

- Index images
- Intensity images
- Binary images
- RGB images

This section discusses how MATLAB and the Image Processing Toolbox represent each of these image types.

Indexed Images

An indexed image consists of a data matrix, `X`, and a colormap matrix, `map`. The data matrix can be of class `uint8`, `uint16`, or `double`. The colormap matrix is an m-by-3 array of class `double` containing floating-point values in the range [0,1]. Each row of `map` specifies the red, green, and blue components of a single color. An indexed image uses direct mapping of pixel values to colormap values. The color of each image pixel is determined by using the corresponding value of `X` as an index into `map`. The value 1 points to the first row in `map`, the value 2 points to the second row, and so on.

A colormap is often stored with an indexed image and is automatically loaded with the image when you use the `imread` function. However, you are not limited to using the default colormap—you can use any colormap that you choose. The figure below illustrates the structure of an indexed image. The pixels in the image are represented by integers, which are pointers (indices) to color values stored in the colormap.

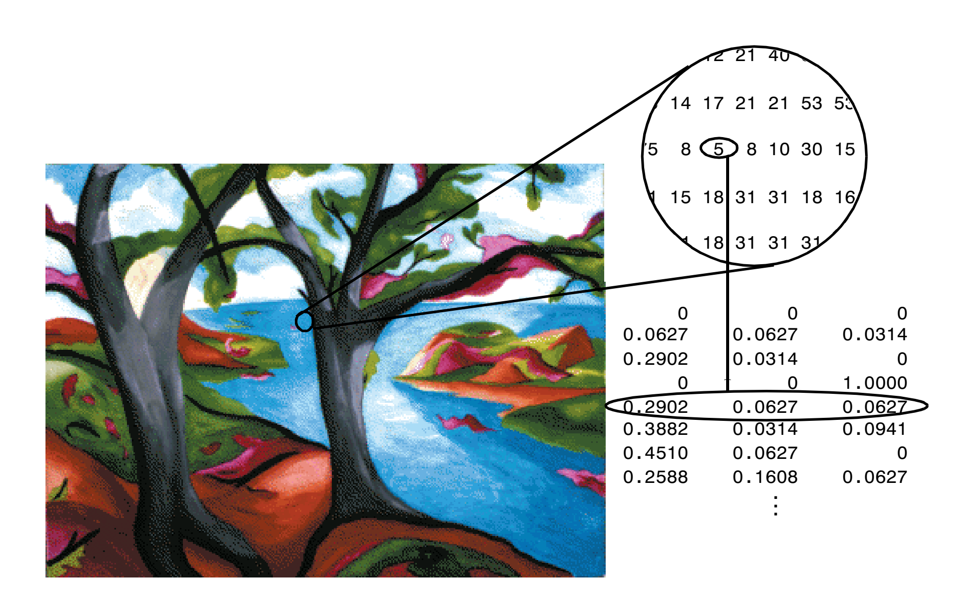


Figure 2-1: Pixel Values Are Indices to a Colormap in Indexed Images

Class and Colormap Offsets

The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8` or `uint16`, there is an offset—the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

The offset is also used in graphics file formats to maximize the number of colors that can be supported. In the image above, the image matrix is of class `double`. Because there is no offset, the value 5 points to the fifth row of the colormap.

Limitations to uint16 Support

Note that the toolbox provides limited support for indexed images of class `uint16`. You can read these images into MATLAB and display them, but before you can process a `uint16` indexed image you must first convert it to either a `double` or a `uint8`. To convert to a `double`, call `im2double`; to reduce the image

to 256 colors or fewer (`uint8`) call `imapprox`. For more information, see the reference pages for `im2double` and `imapprox`.

Intensity Images

An intensity image is a data matrix, `I`, whose values represent intensities within some range. MATLAB stores an intensity image as a single matrix, with each element of the matrix corresponding to one image pixel. The matrix can be of class `double`, `uint8`, or `uint16`. While intensity images are rarely saved with a colormap, MATLAB uses a colormap to display them.

The elements in the intensity matrix represent various intensities, or gray levels, where the intensity 0 usually represents black and the intensity 1, 255, or 65535 usually represents full intensity, or white.

The figure below depicts an intensity image of class `double`.

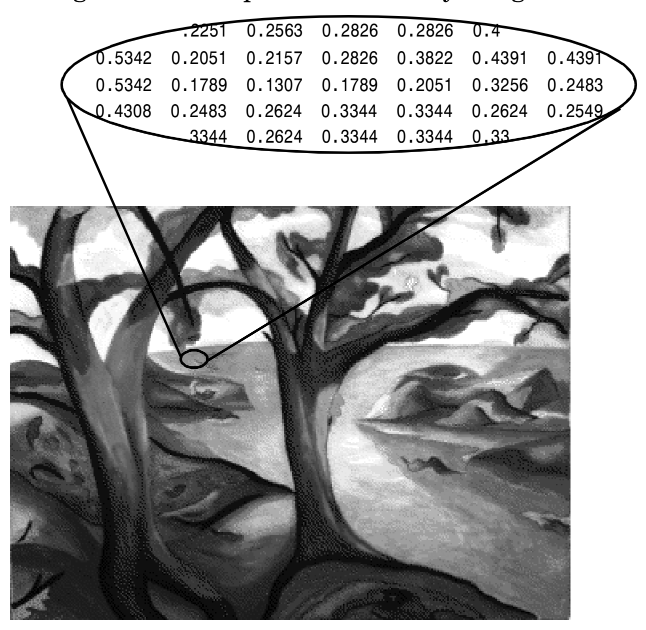


Figure 2-2: Pixel Values in an Intensity Image Define Gray Levels

Binary Images

In a binary image, each pixel assumes one of only two discrete values. Essentially, these two values correspond to on and off. A binary image is stored as a logical array of 0's (off pixels) and 1's (on pixels).

The figure below depicts a binary image.



Figure 2-3: Pixels in a Binary Image Have Two Possible Values: 0 or 1

RGB Images

An RGB image, sometimes referred to as a *truecolor* image, is stored in MATLAB as an m-by-n-by-3 data array that defines red, green, and blue color components for each individual pixel. RGB images do not use a palette. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location. Graphics file formats store RGB images as 24-bit images, where the red, green, and blue components are 8 bits each. This yields a potential of 16 million colors. The precision with which a real-life image can be replicated has led to the commonly used term truecolor image.

An RGB MATLAB array can be of class double, uint8, or uint16. In an RGB array of class double, each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) displays as black, and a pixel whose color components are (1,1,1) displays as white. The three color components for each

pixel are stored along the third dimension of the data array. For example, the red, green, and blue color components of the pixel (10,5) are stored in `RGB(10,5,1)`, `RGB(10,5,2)`, and `RGB(10,5,3)`, respectively.

Figure 2-4 depicts an RGB image of class `double`.

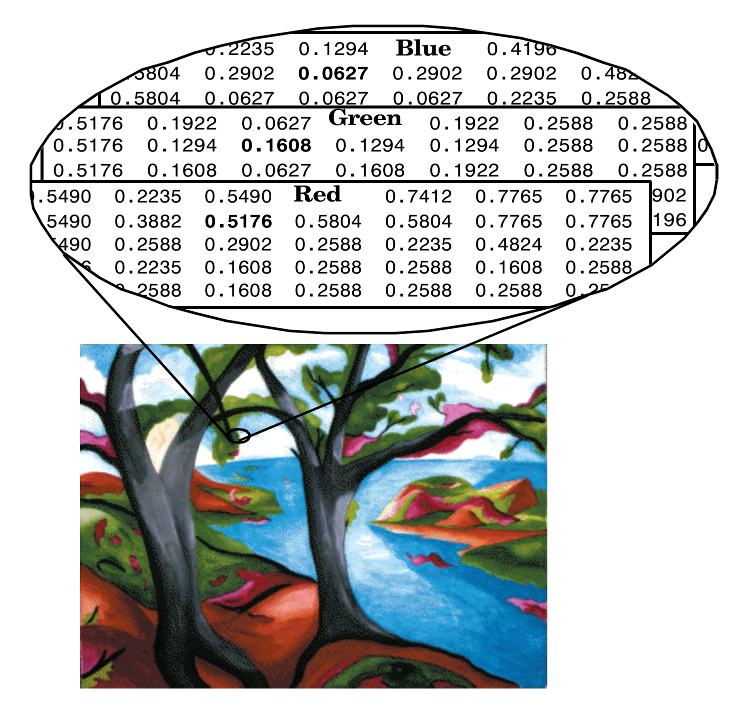


Figure 2-4: The Color Planes of an RGB Image

To determine the color of the pixel at (2,3), you would look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value `0.5176`, (2,3,2) contains `0.1608`, and (2,3,3) contains `0.0627`. The color for the pixel at (2,3) is

```
0.5176 0.1608 0.0627
```

To further illustrate the concept of the three separate color planes used in an RGB image, the code sample below creates a simple RGB image containing uninterrupted areas of red, green, and blue, and then creates one image for each of its separate color planes (red, green, and blue). It displays each color plane image separately, and also displays the original image.

```
RGB=reshape(ones(64,1)*reshape(jet(64),1,192),[64,64,3]);
R=RGB(:,:,1);
G=RGB(:,:,2);
B=RGB(:,:,3);
imshow(R)
figure, imshow(G)
figure, imshow(B)
figure, imshow(RGB)
```

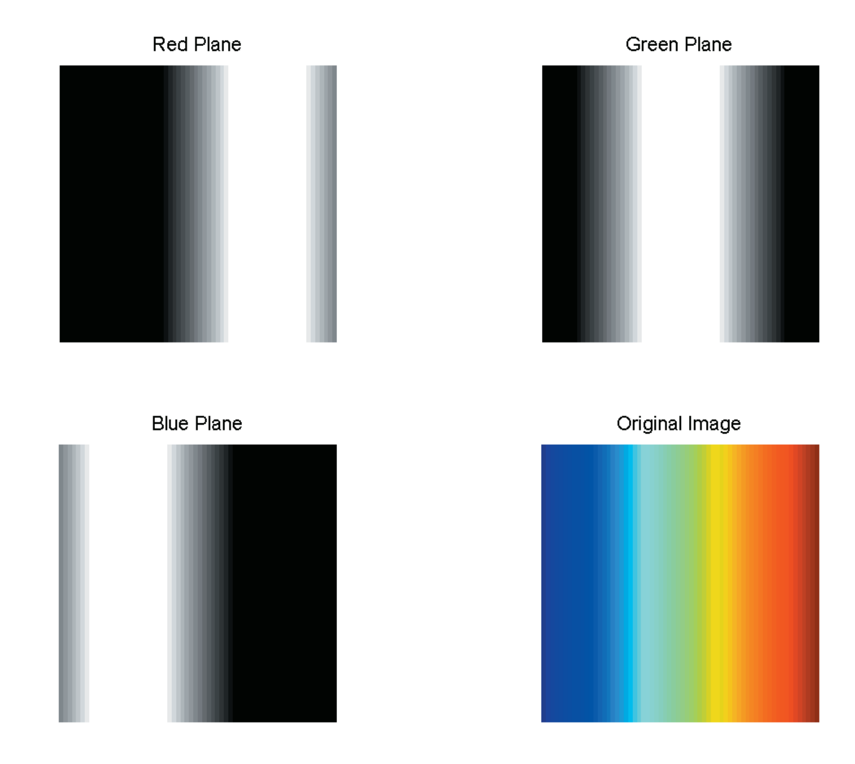


Figure 2-5: The Separated Color Planes of an RGB Image

Notice that each separated color plane in the figure contains an area of white. The white corresponds to the highest values (purest shades) of each separate color. For example, in the Red Plane image, the white represents the highest concentration of pure red values. As red becomes mixed with green or blue, gray pixels appear. The black region in the image shows pixel values that contain no red values, i.e., `R == 0`.

Multiframe Image Arrays

For some applications, you may need to work with collections of images related by time or view, such as magnetic resonance imaging (MRI) slices or movie frames.

The Image Processing Toolbox provides support for storing multiple images in the same array. Each separate image is called a *frame*. If an array holds multiple frames, they are concatenated along the fourth dimension. For example, an array with five 400-by-300 RGB images would be 400-by-300-by-3-by-5. A similar multiframe intensity or indexed image would be 400-by-300-by-1-by-5.

Use the `cat` command to store separate images into one multiframe file. For example, if you have a group of images `A1`, `A2`, `A3`, `A4`, and `A5`, you can store them in a single array using

```
A = cat(4,A1,A2,A3,A4,A5)
```

You can also extract frames from a multiframe image. For example, if you have a multiframe image `MULTI`, this command extracts the third frame.

```
FRM3 = MULTI(:,:,:,3)
```

Note that in a multiframe image array, each image must be the same size and have the same number of planes. In a multiframe indexed image, each image must also use the same colormap.

Multiframe Support Limitations

Many of the functions in the toolbox operate only on the first two or first three dimensions. You can still use four-dimensional arrays with these functions, but you must process each frame individually. For example, this call displays the seventh frame in the array `MULTI`.

```
imshow(MULTI(:,:,:,7))
```

If you pass an array to a function and the array has more dimensions than the function is designed to operate on, your results may be unpredictable. In some cases, the function simply processes the first frame of the array, but in other cases the operation does not produce meaningful results.

See the reference pages for information about how individual functions work with the dimensions of an image array. For more information about displaying multiframe images, see Chapter 3, "Displaying and Printing Images."

Summary of Image Types and Numeric Classes

This table summarizes the way MATLAB interprets data matrix elements as pixel colors, depending on the image type and storage class.

Image Type	Storage Class	Interpretation
Binary	`logical`	An array of zeros (0) and ones (1)
Indexed[1]	`double`	An array of integers in the range [1, p]
	`uint8` or `uint16`	An array of integers in the range [0, p-1]
Intensity[1]	`double`	An array of floating-point values. The typical range of values is [0, 1].
	`uint8` or `uint16`	An array of integers. The typical range of values is [0, 255] or [0, 65535].
RGB (Truecolor)	`double`	An m-by-n-by-3 array of floating-point values in the range [0, 1].
	`uint8` or `uint16`	An m-by-n-by-3 array of integers in the range [0, 255] or [0, 65535].

1. The associated colormap is a p-by-3 array of floating-point values in the range [0, 1]. For intensity images the colormap is typically grayscale.

Converting Image Types

For certain operations, it is helpful to convert an image to a different image type. For example, if you want to filter a color image that is stored as an

indexed image, you should first convert it to RGB format. When you apply the filter to the RGB image, MATLAB filters the intensity values in the image, as is appropriate. If you attempt to filter the indexed image, MATLAB simply applies the filter to the indices in the indexed image matrix, and the results may not be meaningful.

Note When you convert an image from one format to another, the resulting image may look different from the original. For example, if you convert a color indexed image to an intensity image, the resulting image is grayscale, not color. For more information about how the image conversion functions listed below, see their reference pages.

The following table lists all the image conversion functions in the Image Processing Toolbox.

Function	Description
`dither`	Create a binary image from a grayscale intensity image by dithering; create an indexed image from an RGB image by dithering
`gray2ind`	Create an indexed image from a grayscale intensity image
`grayslice`	Create an indexed image from a grayscale intensity image by thresholding
`im2bw`	Create a binary image from an intensity image, indexed image, or RGB image, based on a luminance threshold
`ind2gray`	Create a grayscale intensity image from an indexed image
`ind2rgb`	Create an RGB image from an indexed image
`mat2gray`	Create a grayscale intensity image from data in a matrix, by scaling the data

Function	Description
`rgb2gray`	Create a grayscale intensity image from an RGB image
`rgb2ind`	Create an indexed image from an RGB image

You can also perform certain conversions just using MATLAB syntax. For example, you can convert an intensity image to RGB format by concatenating three copies of the original matrix along the third dimension.

```
RGB = cat(3,I,I,I);
```

The resulting RGB image has identical matrices for the red, green, and blue planes, so the image displays as shades of gray.

In addition to these standard conversion tools, there are some functions that return a different image type as part of the operation they perform. For example, the region-of-interest routines each return a binary image that you can use to mask an indexed or intensity image for filtering or for other operations.

Color Space Conversions

The Image Processing Toolbox represents colors as RGB values, either directly (in an RGB image) or indirectly (in an indexed image). However, there are other methods for representing colors. For example, a color can be represented by its hue, saturation, and value components (HSV). Different methods for representing colors are called *color spaces*.

The toolbox provides a set of routines for converting between RGB and other color spaces. The image processing functions themselves assume all color data is RGB, but you can process an image that uses a different color space by first converting it to RGB, and then converting the processed image back to the original color space. For more information about color space conversion routines, see Chapter 13, “Color.”

Working with Image Data

Images are most commonly stored in MATLAB using the `logical`, `uint8`, `uint16` and `double` data types. This section describes how to work with image data in MATLAB. Topics include:

- Reading images into the MATLAB workspace
- Writing images to files, in many standard graphics file formats
- Querying graphics image files for information stored in header fields
- Converting images between image storage classes
- Converting images between graphics file formats

For information about displaying and printing images, see "Displaying and Printing Images".

You can also perform many standard MATLAB array manipulations on `uint8` and `uint16` image data, including:

- Indexing, including logical indexing
- Reshaping, reordering, and concatenating
- Reading from and writing to MAT-files,
- Using relational operators

Certain MATLAB functions, including the `find`, `all`, `any`, `conv2`, `convn`, `fft2`, `fftn`, and `sum` functions accept `uint8` or `uint16` data but return data in double precision format.

The basic MATLAB arithmetic operators, however, do not accept `uint8` or `uint16` data. For example, if you attempt to add two `uint8` images, `A` and `B`, you get an error, such as,

```
C = A + B
??? Function '+' not defined for variables of class 'uint8'.
```

Because these arithmetic operations are an important part of many image-processing operations, the Image Processing Toolbox includes functions that support these operations on `uint8` and `uint16` data, as well as the other numeric data types. See "Image Arithmetic" for more information.

Reading a Graphics Image

The function `imread` reads an image from any supported graphics image file format, in any of the supported bit depths. Most image file formats use 8 bits to store pixel values. When these are read into memory, MATLAB stores them as class `uint8`. For file formats that support 16-bit data, PNG and TIFF, MATLAB stores the images as class `uint16`. As with MATLAB-generated images, once a image is displayed, it becomes a Handle Graphics Image object.

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

For example, this code reads the image `ngc6543a.jpg` into the MATLAB workspace as the variable `RGB`.

```
RGB = imread('ngc6543a.jpg');
```

In this example, `imread` infers the file format to use from the contents of the file. You can also specify the file format as an argument to `imread`. MATLAB supports the following graphics file formats:

- BMP (Microsoft Windows Bitmap)
- CUR (Microsoft Windows Cursor resource)
- GIF (Graphics Interchange Format)
- HDF (Hierarchical Data Format)
- ICO (Windows Icon resource
- JPEG (Joint Photographic Experts Group)
- PBM (Portable Bitmap)
- PCX (Windows Paintbrush)
- PGM (Portable Graymap)
- PNG (Portable Network Graphics)
- PPM (Portable Pixmap)
- RAS (Sun Raster image)
- TIFF (Tagged Image File Format)
- XWD (X Window Dump)

For the latest information concerning the bit depths and/or image formats supported, see the reference pages for imread.

Writing a Graphics Image

The function imwrite writes an image to a graphics file in one of the supported formats. The most basic syntax for imwrite takes the image variable name and a filename. If you include an extension in the filename, MATLAB infers the desired file format from it. This example loads an image of a clown from a MAT-file, and then creates a BMP file containing the clown image.

```
load clown
whos
  Name          Size          Bytes  Class

  X           200x320        512000  double array
  caption       2x1               4  char array
  map          81x3            1944  double array

Grand total is 64245 elements using 513948 bytes

imwrite(X,map,'clown.bmp')
```

For some graphics formats, you can specify additional parameters. One of the additional parameters for PNG files sets the bit depth. This example writes an intensity image I to a 4-bit PNG file.

```
imwrite(I,'clown.png','BitDepth',4);
```

The bit depths and image types supported for each format are shown in the reference pages for imwrite.

This example writes an image A to a JPEG file with a compression quality setting of 100 (the default is 75).

```
imwrite(A, 'myfile.jpg', 'Quality', 100);
```

Output File Storage Classes

`imwrite` uses the following rules to determine the storage class used in the output image.

Storage Class of Image	Storage Class of Output Image File
`logical`	If the output image file format specified supports 1-bit images, `imwrite` creates a 1-bit image file. If the output image file format specified *does not* support 1-bit images, such as JPEG, `imwrite` converts the image to a class `uint8` intensity image.
`uint8`	If the output image file format specified supports 8-bit images, `imwrite` creates an 8-bit image file.
`uint16`	If the output image file format specified supports 16-bit images (PNG or TIFF), `imwrite` creates a 16-bit image file. If the output image file format specified *does not* support 16-bit images, `imwrite` scales the image data to class `uint8` and creates an 8-bit image file.
`double`	MATLAB scales the image data to uint8 and creates an 8-bit image file because most image file formats use 8-bits.

See the reference entry for `imwrite` for more information.

Querying a Graphics File

The `imfinfo` function enables you to obtain information about graphics files that are in any of the formats supported by the toolbox. The information you obtain depends on the type of file, but it always includes at least the following:

- Name of the file
- File format
- Version number of the file format

- File modification date
- File size in bytes
- Image width in pixels
- Image height in pixels
- Number of bits per pixel
- Image type: RGB (truecolor), intensity (grayscale), or indexed

See the reference entry for `imfinfo` for more information.

Converting Image Storage Classes

You can convert `uint8` and `uint16` data to double precision using the MATLAB function, `double`. However, converting between storage classes changes the way MATLAB and the toolbox interpret the image data. If you want the resulting array to be interpreted properly as image data, you need to rescale or offset the data when you convert it.

For easier conversion of storage classes, use one of these toolbox functions: `im2double`, `im2uint8`, and `im2uint16`. These functions automatically handle the rescaling and offsetting of the original data. For example, this command converts a double-precision RGB image with data in the range [0,1] to a `uint8` RGB image with data in the range [0,255].

```
RGB2 = im2uint8(RGB1);
```

Losing Information in Conversions

When you convert to a class that uses fewer bits to represent numbers, you generally lose some of the information in your image. For example, a `uint16` intensity image is capable of storing up to 65,536 distinct shades of gray, but a `uint8` intensity image can store only 256 distinct shades of gray. When you convert a `uint16` intensity image to a `uint8` intensity image, `im2uint8` *quantizes* the gray shades in the original image. In other words, all values from 0 to 127 in the original image become 0 in the `uint8` image, values from 128 to 385 all become 1, and so on. This loss of information is often not a problem, however, since 256 still exceeds the number of shades of gray that your eye is likely to discern.

Converting Indexed Images

It is not always possible to convert an indexed image from one storage class to another. In an indexed image, the image matrix contains only indices into a colormap, rather than the color data itself, so no quantization of the color data is possible during the conversion.

For example, a `uint16` or `double` indexed image with 300 colors cannot be converted to `uint8`, because `uint8` arrays have only 256 distinct values. If you want to perform this conversion, you must first reduce the number of the colors in the image using the `imapprox` function. This function performs the quantization on the colors in the colormap, to reduce the number of distinct colors in the image. See "Reducing Colors in an Indexed Image" on page 13-12 for more information.

Converting Graphics File Formats

To change the graphics format of an image, use `imread` to read in the image and then save the image with `imwrite`, specifying the appropriate format.

For example, to convert an image from a BMP to a PNG, read the BMP image using `imread`, convert the storage class if necessary, and then write the image using `imwrite`, with `'PNG'` specified as your target format.

```
bitmap = imread('mybitmap.bmp','bmp');
imwrite(bitmap,'mybitmap.png','png');
```

For the specifics of which bit depths are supported for the different graphics formats, and for how to specify the format type when writing an image to file, see the reference entries for `imread` and `imwrite`.

Image Arithmetic

Image arithmetic is the implementation of standard arithmetic operations, such as addition, subtraction, multiplication, and division, on images. Image arithmetic has many uses in image processing both as a preliminary step in more complex operations and by itself. For example, image subtraction can be used to detect differences between two or more images of the same scene or object.

You can do image arithmetic using the MATLAB arithmetic operators; however, you must convert the images to class `double` to use these operators. To make working with images more convenient, the Image Processing Toolbox includes a set of functions that implement arithmetic operations for all numeric, nonsparse data types. The advantages to using these functions include:

- No conversion to the `double` data type is necessary. The functions accept any numeric data type, including `uint8`, `uint16`, and `double`, and return the result image in the same format. Note that the functions perform the operations in double precision, on an element-by-element basis, but do not convert images to double precision values in the MATLAB workspace.
- Overflow is handled automatically. The functions truncate return values to fit the data type. For details about this truncation, see "Image Arithmetic Truncation Rules" on page 2-22.

See "Summary of Image Arithmetic Functions" on page 2-22 for a complete list. For more information about using these functions to perform arithmetic operations, see these sections:

- "Adding Images" on page 2-23
- "Subtracting Images" on page 2-24
- "Multiplying Images" on page 2-25
- "Dividing Images" on page 2-27
- "Nesting Calls to Image Arithmetic Functions" on page 2-27

Summary of Image Arithmetic Functions

The following table lists the image arithmetic functions. For more complete descriptions, see their reference pages.

Function	Description
imabsdiff	Absolute difference of two images
imadd	Add two images
imcomplement	Complement an image
imdivide	Divide two images
imlincomb	Compute linear combination of two images
immultiply	Multiply two images
imsubtract	Subtract two images

Image Arithmetic Truncation Rules

The results of integer arithmetic can easily overflow the data type allotted for storage. For example, the maximum value you can store in uint8 data is 255. Arithmetic operations can also result in fractional values, which cannot be represented using integer arrays.

The image arithmetic functions use these rules for integer arithmetic:

- Values that exceed the range of the integer type are truncated to that range
- Fractional values are rounded

For example, if the data type is uint8, results greater than 255 (including Inf) are set to 255. The following table lists some additional examples.

Result	Class	Truncated Value
300	uint8	255
-45	uint8	0
10.5	uint8	11

Adding Images

To add two images or add a constant value to an image, use the `imadd` function. `imadd` adds the value of each pixel in one of the input images with the corresponding pixel in the other input image and returns the sum in the corresponding pixel of the output image.

Image addition has many uses in image processing. For example, the following code fragment uses addition to superimpose one image on top of another. The images must be the same size and class.

```
I = imread('rice.tif');
J = imread('cameraman.tif');
K = imadd(I,J);
imshow(K)
```



You can also use addition to brighten an image by adding a constant value to each pixel. For example, the following code brightens an RGB image.

```
RGB = imread('flowers.tif');
RGB2 = imadd(RGB,50);
subplot(1,2,1); imshow(RGB);
subplot(1,2,2); imshow(RGB2);
```



Original Image



Image After Addition

Handling Overflow

When you add the pixel values of two images, the result can easily overflow the maximum value supported by the data type, especially for `uint8` data. When overflow occurs, `imadd` truncates the value to the maximum value supported by the data type. This is an effect known as *saturation*. For example, `imadd` truncates `uint8` data at 255. To avoid saturation, convert the image to a larger data type, such as `uint16`, before performing the addition.

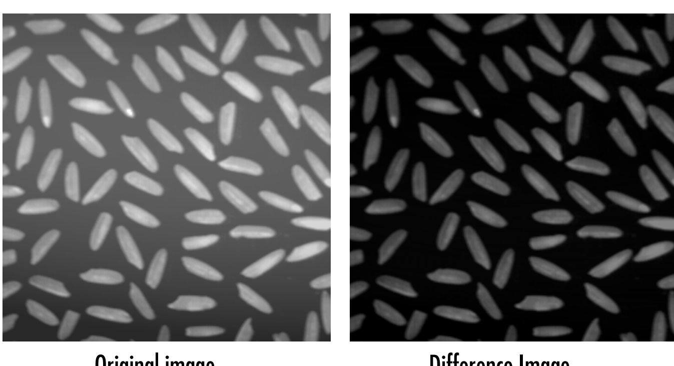
Subtracting Images

To subtract one image from another, or subtract a constant value from an image, use the `imsubtract` function. `imsubtract` subtracts each pixel value in one of the input images from the corresponding pixel in the other input image and returns the result in the corresponding pixel in an output image.

Image subtraction can be used as a preliminary step in more complex image processing or by itself. For example, you can use image subtraction to detect changes in a series of images of the same scene. This code fragment subtracts the background from an image of rice grains. The images must be the same size and class.

```
rice= imread('rice.tif');
background = imopen(rice, strel('disk',15));
```

```
rice2 = imsubtract(rice,background);
imshow(rice),figure,imshow(rice2);
```



Original image   Difference Image

To subtract a constant from each pixel in I, replace Y with a constant, as in the following example.

```
Z = imsubtract(I,50);
```

Handling Negative Values

Subtraction can result in a negative values for certain pixels. When this occurs with unsigned data types, such as uint8 or uint16, the imsubtract function truncates the negative value to zero (0), which displays as black. To avoid negative values, but preserve the value differentiation of these pixels, use the imabsdiff function. The imabsdiff function calculates the absolute difference between each corresponding pixel in the two images so the result is always nonnegative.

Multiplying Images

To multiply two images, use the immultiply function. immultiply does an element-by-element multiplication (.*) of each corresponding pixel in a pair of input images and returns the product of these multiplications in the corresponding pixel in an output image.

Image multiplication by a constant, referred to as *scaling*, is a common image processing operation. When used with a scaling factor greater than one, scaling brightens an image; a factor less than one darkens an image. Scaling generally

produces a much more natural brightening/darkening effect than simply adding an offset to the pixels, since it preserves the relative contrast of the image better. For example, this code scales an image by a constant factor.

```
I = imread('moon.tif');
J = immultiply(I,1.2);
imshow(I);
figure, imshow(J)
```



Original image   Image after multiplication

Handling Overflow

Multiplication of `uint8` images very often results in overflow. The `immultiply` function truncates values that overflow the data type to the maximum value. To avoid truncation, convert `uint8` images to a larger data type, such as `uint16`, before performing multiplication.

Dividing Images

To divide two images, use the imdivide function. The imdivide function does an element-by-element division (./) of each corresponding pixel in a pair of input images. The immultiply function returns the result in the corresponding pixel in an output image.

Image division, like image subtraction, can be used to detect changes in two images. However, instead of giving the absolute change for each pixel, division gives the fractional change or ratio between corresponding pixel values. Image division is also called *ratioing*.

For example, the following code divides the rice grain image by a morphologically opened version of the itself. (For information about morphological image processing, see Chapter 9, “Morphological Operations.”) The images must be the same size and class.

```
I = imread('rice.tif');
background = imopen(I, strel('disk',15));
Ip = imdivide(I,background);
imshow(Ip,[])
```

Nesting Calls to Image Arithmetic Functions

You can use the image arithmetic functions in combination to perform a series of operations. For example, to calculate the average of two images,

$$C = \frac{A + B}{2}$$

You could enter

```
I = imread('rice.tif');
I2 = imread('cameraman.tif');
K = imdivide(imadd(I,I2), 2); % not recommended
```

However, when used with uint8 or uint16 data, each arithmetic function truncates its result before passing it on to the next operation. This truncation can significantly reduce the amount of information in the output image. A better way to perform this series of calculations is to use the imlincomb function. imlincomb performs all the arithmetic operations in the linear combination in double precision and only truncates the final result.

```
K = imlincomb(.5,I,.5,I2); % recommended
```

Coordinate Systems

Locations in an image can be expressed in various coordinate systems, depending on context. This section discusses the two main coordinate systems used in the Image Processing Toolbox, and the relationship between them. These two coordinate systems are described in:

- "Pixel Coordinates"
- "Spatial Coordinates" on page 2-29

Pixel Coordinates

Generally, the most convenient method for expressing locations in an image is to use pixel coordinates. In this coordinate system, the image is treated as a grid of discrete elements, ordered from top to bottom and left to right, as illustrated by the following figure.

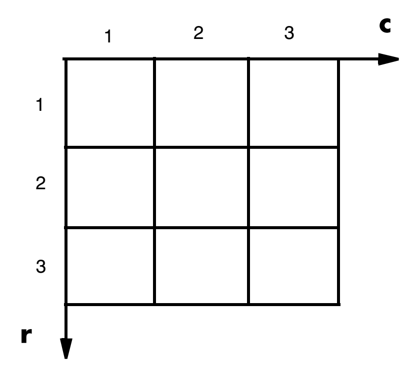


Figure 2-6: The Pixel Coordinate System

For pixel coordinates, the first component `r` (the row) increases downward, while the second component `c` (the column) increases to the right. Pixel coordinates are integer values and range between 1 and the length of the row or column.

There is a one-to-one correspondence between pixel coordinates and the coordinates MATLAB uses for matrix subscripting. This correspondence makes the relationship between an image's data matrix and the way the image

displays easy to understand. For example, the data for the pixel in the fifth row, second column is stored in the matrix element (5,2).

Spatial Coordinates

In the pixel coordinate system, a pixel is treated as a discrete unit, uniquely identified by a single coordinate pair, such as (5,2). From this perspective, a location such as (5.3,2.2) is not meaningful.

At times, however, it is useful to think of a pixel as a square patch. From this perspective, a location such as (5.3,2.2) *is* meaningful, and is distinct from (5,2). In this spatial coordinate system, locations in an image are positions on a plane, and they are described in terms of `x` and `y` (not `r` and `c` as in the pixel coordinate system).

The following figure illustrates the spatial coordinate system used for images. Notice that `y` increases downward.

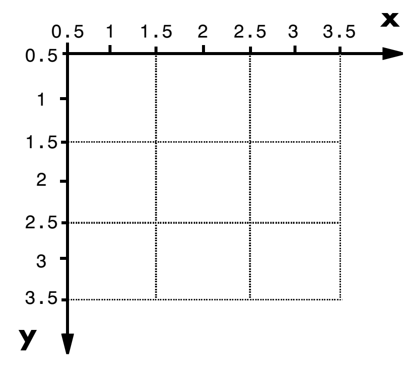


Figure 2-7: The Spatial Coordinate System

This spatial coordinate system corresponds closely to the pixel coordinate system in many ways. For example, the spatial coordinates of the center point of any pixel are identical to the pixel coordinates for that pixel.

There are some important differences, however. In pixel coordinates, the upper-left corner of an image is (1,1), while in spatial coordinates, this location by default is (0.5,0.5). This difference is due to the pixel coordinate system being discrete, while the spatial coordinate system is continuous. Also, the upper-left corner is always (1,1) in pixel coordinates, but you can specify a

nondefault origin for the spatial coordinate system. See "Using a Nondefault Spatial Coordinate System" on page 2-30 for more information.

Another potentially confusing difference is largely a matter of convention: the order of the horizontal and vertical components is reversed in the notation for these two systems. As mentioned earlier, pixel coordinates are expressed as (r,c), while spatial coordinates are expressed as (x,y). In the reference pages, when the syntax for a function uses r and c, it refers to the pixel coordinate system. When the syntax uses x and y, it refers to the spatial coordinate system.

Using a Nondefault Spatial Coordinate System

By default, the spatial coordinates of an image correspond with the pixel coordinates. For example, the center point of the pixel in row 5, column 3 has spatial coordinates x=3, y=5. (Remember, the order of the coordinates is reversed.) This correspondence simplifies many of the toolbox functions considerably. Several functions primarily work with spatial coordinates rather than pixel coordinates, but as long as you are using the default spatial coordinate system, you can specify locations in pixel coordinates.

In some situations, however, you may want to use a nondefault spatial coordinate system. For example, you could specify that the upper-left corner of an image is the point (19.0,7.5), rather than (0.5,0.5). If you call a function that returns coordinates for this image, the coordinates returned will be values in this nondefault spatial coordinate system.

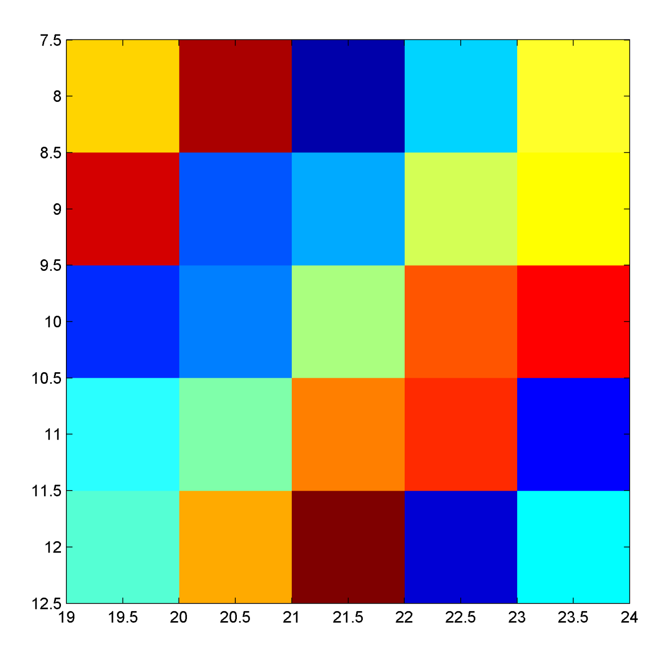
To establish a nondefault spatial coordinate system, you can specify the XData and YData image properties when you display the image. These properties are two-element vectors that control the range of coordinates spanned by the image. By default, for an image A, XData is [1 size(A,2)], and YData is [1 size(A,1)].

For example, if A is a 100 row by 200 column image, the default XData is [1 200], and the default YData is [1 100]. The values in these vectors are actually the coordinates for the center points of the first and last pixels (not the pixel edges), so the actual coordinate range spanned is slightly larger; for instance, if XData is [1 200], the *x*-axis range spanned by the image is [0.5 200.5].

These commands display an image using nondefault XData and YData.

```
A = magic(5);
```

```
x = [19.5 23.5];
y = [8.0 12.0];
image(A,'XData',x,'YData',y), axis image, colormap(jet(25))
```



For information about the syntax variations that specify nondefault spatial coordinates, see the reference page for `imshow`.

3

Displaying and Printing Images

This section introduces the image display techniques supported by the Image Processing Toolbox for each image type supported by the toolbox (e.g., RGB, intensity). Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Color approximation	There are two ways in which this term is used in MATLAB: • The method by which MATLAB chooses the best colors for an image whose number of colors you are decreasing • The automatic choice of screen colors MATLAB makes when displaying on a system with limited color display capability
Screen bit depth	The number of bits per screen pixel
Screen color resolution	The number of distinct colors that can be produced by the screen

Displaying Images

In MATLAB, the primary way to display images is by using the image function. This function creates a Handle Graphics® image object, and it includes syntax for setting the various properties of the object. MATLAB also includes the imagesc function, which is similar to image but which automatically scales the input data.

The Image Processing Toolbox includes an additional display routine called imshow. Like image and imagesc, this function creates a Handle Graphics image object. However, imshow also automatically sets various Handle Graphics properties and attributes of the image to optimize the display.

This section discusses displaying images using imshow. In general, using imshow for image processing applications is preferable to using image and imagesc. It is easier to use and in most cases, displays an image using one image pixel per screen pixel. (For more information about image and imagesc, see their pages in the MATLAB Function Reference or see the MATLAB graphics documentation.)

Note One of the most common toolbox usage errors is using the wrong syntax of imshow for your image type. This chapter shows which syntax is appropriate for each type of image. If you need help determining what type of image you are working with, see "Image Types in the Toolbox" on page 2-5.

Displaying Indexed Images

To display an indexed image with imshow, specify both the image matrix and the colormap.

```
imshow(X,map)
```

For each pixel in X, imshow displays the color stored in the corresponding row of map. The relationship between the values in the image matrix and the colormap depends on whether the image matrix is of class double, uint16, or uint8. If the image matrix is of class double, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class uint8 or uint16, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on. (The

offset is handled automatically by the image object, and is not controlled through a Handle Graphics property.)

`imshow` directly maps each pixel in an indexed image to its corresponding colormap entry. If the colormap contains a greater number of colors than the image, `imshow` ignores the extra colors in the colormap. If the colormap contains fewer colors than the image requires, `imshow` sets all image pixels over the limits of the colormap's capacity to the last color in the colormap. For example, if an image of class `uint8` contains 256 colors, and you display it with a colormap that contains only 16 colors, all pixels with a value of 15 or higher are displayed with the last color in the colormap.

The Image and Axes Properties of an Indexed Image

When you display an indexed image, `imshow` sets the Handle Graphics properties that control how colors display, as follows:

- The image `CData` property is set to the data in `X`.
- The image `CDataMapping` property is set to `direct`.
- The axes `CLim` property does not apply, because `CDataMapping` is set to `direct`.
- The figure `Colormap` property is set to the data in `map`.

Displaying Intensity Images

To display a intensity (grayscale) image, the most basic syntax is

```
imshow(I)
```

`imshow` displays the image by *scaling* the intensity values to serve as indices into a grayscale colormap. If `I` is `double`, a pixel value of 0.0 is displayed as black, a pixel value of 1.0 is displayed as white, and pixel values in between are displayed as shades of gray. If `I` is `uint8`, then a pixel value of 255 is displayed as white. If I is `uint16`, then a pixel value of 65535 is displayed as white.

Intensity images are similar to indexed images in that each uses an m-by-3 RGB colormap, but normally, you will not specify a colormap for an intensity image. MATLAB displays intensity images by using a grayscale system colormap (where R=G=B). By default, the number of levels of gray in the colormap is 256 on systems with 24-bit color, and 64 or 32 on other systems. (See "Working with Different Screen Bit Depths" on page 13-3 for a detailed explanation.)

Another syntax form of `imshow` for intensity images enables you to explicitly specify the number of gray levels to use. To display an image `I` with 32 gray levels, specify a value for `n`.

```
imshow(I,32)
```

Because MATLAB scales intensity images to fill the colormap range, a colormap of any size can be used. Larger colormaps enable you to see more detail, but they also use up more color slots. The availability of color slots is discussed further in “Displaying Multiple Images” on page 3-17, and also in “Working with Different Screen Bit Depths” on page 13-3.

Displaying Intensity Images That Have Unconventional Ranges

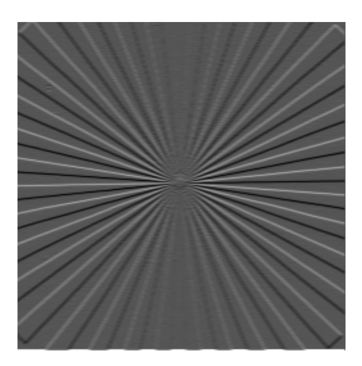
In some cases, you may have data you want to display as an intensity image, even though the data is outside the conventional toolbox range (i.e., [0,1] for `double` arrays, [0,255] for `uint8` arrays, or [0,65535] for `uint16` arrays). For example, if you filter an intensity image, some of the output data may fall outside the range of the original data.

To display unconventional range data as an image, you can specify the data range directly, using

```
imshow(I,[low high])
```

If you use an empty matrix (`[]`) for the data range, `imshow` scales the data automatically, setting `low` and `high` to the minimum and maximum values in the array. The next example filters an intensity image, creating unconventional range data. `imshow` is then called using an empty matrix.

```
I = imread('testpat1.tif');
J = filter2([1 2;-1 -2],I);
imshow(J,[]);
```



When you use this syntax, imshow sets the axes CLim property to [min(J(:)) max(J(:))]. CDataMapping is always scaled for intensity images, so that the value min(J(:)) is displayed using the first colormap color, and the value max(J(:)) is displayed using the last colormap color.

The Image and Axes Properties of an Intensity Image

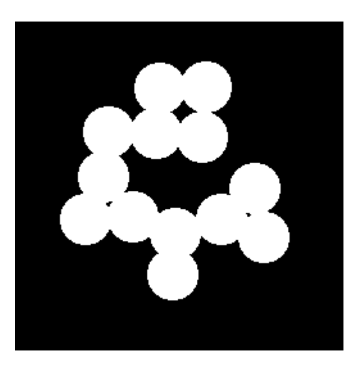
When you display an intensity image, imshow sets the Handle Graphics properties that control how colors display, as follows:

- The image CData property is set to the data in I.
- The image CDataMapping property is set to scaled.
- The axes CLim property is set to [0 1] if the image matrix is of class double, [0 255] if the matrix is of class uint8, or [0 65535] if it is of class uint16.
- The figure Colormap property is set to a grayscale colormap whose values range from black to white.

Displaying Binary Images

To display a binary image, the syntax is

```
BW = imread('circles.tif');
imshow(BW)
```



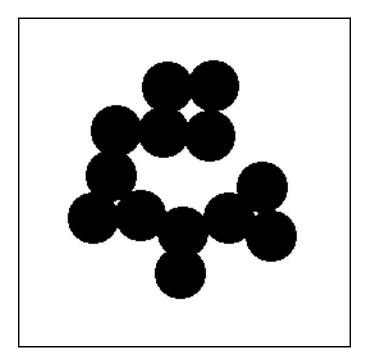
In MATLAB, a binary image is of class `logical`. Binary images contain only 0's and 1's. Pixels with the value 0 display as black; pixels with the value 1 display as white.

Note For the toolbox to interpret the image as binary, it must be of class `logical`. Intensity images that happen to contain only 0's and 1's are not binary images.

Changing the Display Colors of a Binary Image

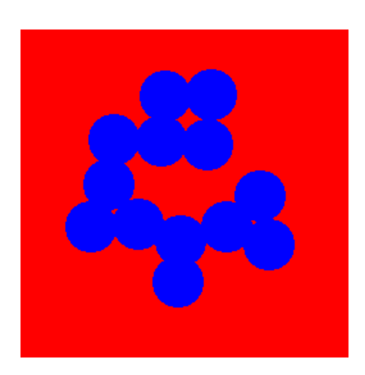
You may prefer to invert binary images when you display them, so that 0 values display as white and 1 values display as black. To do this, use the NOT (~) operator in MATLAB. (In this example, a box is drawn around the image to show the image boundary.) For example,

```
imshow(~BW)
```



You can also display a binary image using a colormap. For example, the following command displays 0's as red and 1's as blue.

```
imshow(BW,[1 0 0; 0 0 1])
```



Reading and Writing Binary Images

In certain file formats, a binary image can be stored in a 1-bit format. When you read in a binary image in 1-bit format, MATLAB represents it in the workspace as a `logical` array.

By default, MATLAB writes binary images as 1-bit images, if the file format supports it.

```
imwrite(BW,'test.tif'); % MATLAB supports writing 1-bit TIFFs.
```

To verify the bit depth of `test.tif`, call `imfinfo`. As you will see, the `BitDepth` field indicates that it has been saved as a 1-bit image, with the beginning of your output looking something like this.

```
imfinfo('test.tif')
ans =
                  Filename: 'd:\mystuff\grid.tif'
               FileModDate: '25-Nov-1998 11:36:17'
                  FileSize: 340
                    Format: 'tif'
             FormatVersion: []
                     Width: 20
                    Height: 20
                  BitDepth: 1
                 ColorType: 'grayscale'
           FormatSignature: [73 73 42 0]
                 ByteOrder: 'little-endian'
            NewSubfileType: 0
             BitsPerSample: 1
             Compression: 'CCITT 1D'
                ...
```

Note You may have noticed that the `ColorType` field of the binary image queried above has a value of `'grayscale'`. MATLAB sets this field to one of three values: `'grayscale'`, `'indexed'`, and `'truecolor'`. When reading an image, MATLAB evaluates the image type by checking both the `BitDepth` and the `ColorType` fields.

The Image and Axes Properties of a Binary Image

`imshow` sets the Handle Graphics properties that control how colors display, as follows:

- The image `CData` is set to the data in `BW`.
- The image `CDataMapping` property is set to `direct`.
- The axes `CLim` property is set to [0 1].
- The figure `Colormap` property is set to a grayscale colormap whose values range from black to white.

Displaying RGB Images

RGB images, also called *truecolor* images, represent color values directly, rather than through a colormap.

To display an RGB image, the most basic syntax is

```
imshow(RGB)
```

`RGB` is m-by-n-by-3 array. For each pixel (`r,c`) in `RGB`, `imshow` displays the color represented by the triplet (`r,c,1:3`).

Systems that use 24 bits per screen pixel can display truecolor images directly, because they allocate 8 bits (256 levels) each to the red, green, and blue color planes. On systems with fewer colors, MATLAB displays the image using a combination of color approximation and dithering. See "Working with Different Screen Bit Depths" on page 13-3 for more information.

The Image and Axes Properties of an RGB Image

When you display an RGB image, `imshow` sets the Handle Graphics properties that control how colors display, as follows:

- The image `CData` property is set to the data in `RGB`. The data will be three-dimensional. When `CData` is three-dimensional, MATLAB interprets the array as truecolor data, and ignores the values of the `CDataMapping`, `CLim`, and `Colormap` properties.
- The image `CDataMapping` property is ignored.
- The axes `CLim` property is ignored.
- The figure `Colormap` property is ignored.

Displaying Images Directly from Disk

Generally, when you want to display an image, you will first use `imread` to load it and the data is stored as one or more variables in the MATLAB workspace. However, if you do not want to load an image directly before displaying it, you can display a file directly using this syntax.

```
imshow filename
```

The file must be in the current directory or on the MATLAB path.

For example, to display a file named `flowers.tif`,

```
imshow flowers.tif
```

If the image has multiple frames, `imshow` will only display the first frame. For information on the display options for multiframe images, see "Displaying Multiframe Images" on page 3-13.

This syntax is very useful for scanning through images. Note, however, that when you use this syntax, the image data is not stored in the MATLAB workspace. If you want to bring the image into the workspace, use the `getimage` function, which gets the image data from the current Handle Graphics image object. For example,

```
rgb = getimage;
```

will assign `flowers.tif` to `rgb` if the figure window in which it is displayed is currently active.

Special Display Techniques

In addition to `imshow`, the toolbox includes functions that perform specialized display operations, or exercise more direct control over the display format. These functions, together with the MATLAB graphics functions, provide a range of image display options.

This section includes the following topics:

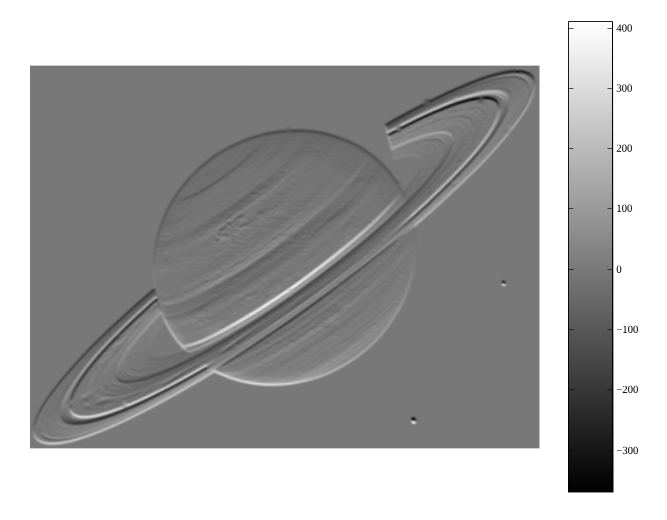
Adding a Colorbar

Use the `colorbar` function to add a colorbar to an axes object. If you add a colorbar to an axes object that contains an image object, the colorbar indicates the data values that the different colors in the image correspond to.

Seeing the correspondence between data values and the colors displayed by using a colorbar is especially useful if you are displaying unconventional range data as an image, as described under "Displaying Intensity Images That Have Unconventional Ranges" on page 3-5.

In the example below, a grayscale image of class `uint8` is filtered, resulting in data that is no longer in the range [0,255].

```
I = imread('saturn.tif');
h = [1 2 1; 0 0 0; -1 -2 -1];
I2 = filter2(h,I);
imshow(I2,[]), colorbar
```



Displaying Multiframe Images

A multiframe image is an image file that contains more than one image. The MATLAB-supported formats that enable the reading and writing of multiframe images are HDF and TIFF. See “Multiframe Image Arrays” on page 2-11 for more information about reading and writing multiframe images.

Once read into MATLAB, the image frames of a multiframe image are always handled in the fourth dimension. Multiframe images can be loaded from disk using a special syntax of `imread`, or created using MATLAB. Multiframe images can be displayed in several different ways; to display a multiframe image, you can

- Display the frames individually, using the `imshow` function. See “Displaying the Frames of a Multiframe Image Individually” on page 3-14 below.
- Display all of the frames at once, using the `montage` function. See “Displaying All Frames of a Multiframe Image at Once” on page 3-15.
- Convert the frames to a movie, using the `immovie` function. See “Converting a Multiframe Image to a Movie” on page 3-16.

Displaying the Frames of a Multiframe Image Individually

In MATLAB, the frames of a multiframe image are handled in the fourth dimension. To view an individual frame, call `imshow` and specify the frame using standard MATLAB indexing notation. For example, to view the seventh frame in the intensity array `I`,

```
imshow(I(:,:,:,7))
```

The following example loads `mri.tif` and displays the third frame.

```
% Initialize an array to hold the 27 frames of mri.tif
mri = uint8(zeros(128,128,1,27));
for frame=1:27
   % Read each frame into the appropriate frame in memory
   [mri(:,:,:,frame),map] = imread('mri.tif',frame);
end
imshow(mri(:,:,:,3),map);
```



Intensity, indexed, and binary multiframe images have a dimension of m-by-n-by-1-by-k, where k represents the total number of frames, and 1 signifies that the image data has just one color plane. Therefore, the following call,

```
imshow(mri(:,:,:,3),map);
```

is equivalent to,

```
imshow(mri(:,:,1,3),map);
```

RGB multiframe images have a dimension of m-by-n-by-3-by-k, where k represents the total number of frames, and 3 signifies the existence of the three color planes used in RGB images. This example,

```
imshow(RGB(:,:,:,7));
```

shows all three color planes of the seventh frame, and is *not* equivalent to

```
imshow(RGB(:,:,3,7));
```

which shows only the third color plane (blue) of the seventh frame. These two calls will only yield the same results if the image is RGB grayscale (R=G=B).

Displaying All Frames of a Multiframe Image at Once

To view all of the frames in a multiframe array at one time, use the `montage` function. `montage` divides a figure into multiple display regions and displays each image in a separate region.

The syntax for `montage` is similar to the `imshow` syntax. To display a multiframe intensity image, the syntax is

```
montage(I)
```

To display a multiframe indexed image, the syntax is

```
montage(X,map)
```

Note All of the frames in a multiframe indexed array must use the same colormap.

This example loads and displays all frames of a multiframe indexed image.

```
% Initialize an array to hold the 27 frames of mri.tif.
mri = uint8(zeros(128,128,1,27));
for frame=1:27
   % Read each frame into the appropriate frame in memory.
   [mri(:,:,:,frame),map] = imread('mri.tif',frame);
end
montage(mri,map);
```

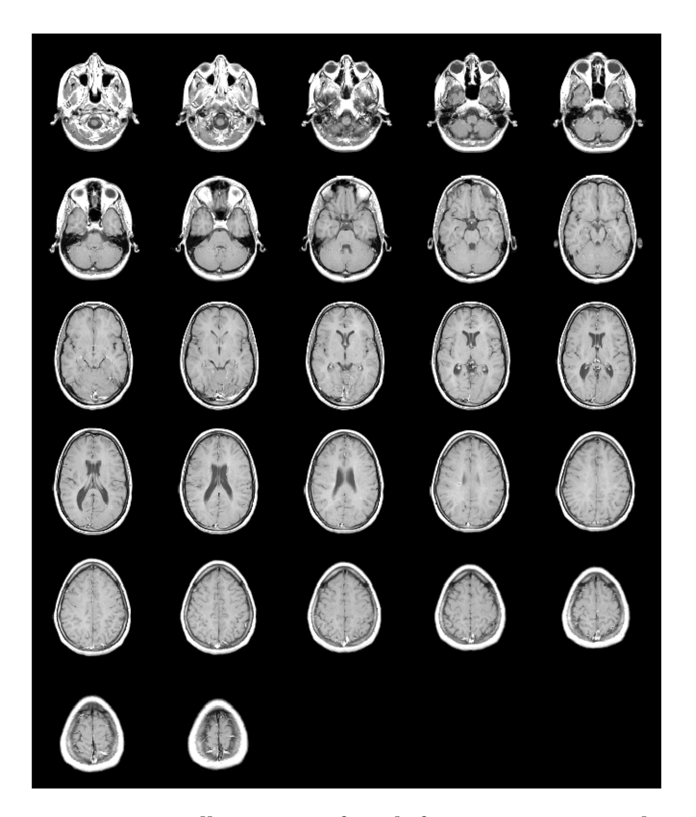


Figure 3-1: All Frames of Multiframe Image Displayed in One Figure

Notice that `montage` displays images in a row-wise manner. The first frame appears in the first position of the first row, the next frame in the second position of the first row, and so on. `montage` arranges the frames so that they roughly form a square.

Converting a Multiframe Image to a Movie

To create a MATLAB movie from a multiframe image array, use the `immovie` function.

This call creates a movie from a multiframe indexed image `X`

```
mov = immovie(X,map);
```

where X is a four-dimensional array of images that you want to use for the movie.

You can play the movie in MATLAB using the movie function.

```
movie(mov);
```

This example loads the multiframe image mri.tif and makes a movie out of it. It won't do any good to show the results here, so try it out; it's fun to watch.

```
% Initialize and array to hold the 27 frames of mri.tif.
mri = uint8(zeros(128,128,1,27));
for frame=1:27
   % Read each frame into the appropriate frame in memory.
   [mri(:,:,:,frame),map] = imread('mri.tif',frame);
end

mov = immovie(mri,map);
movie(mov);
```

Note that immovie displays the movie as it is being created, so you will actually see the movie twice. The movie runs much faster the second time (using movie).

Note MATLAB movies require MATLAB in order to be run. To make a movie that can be run outside of MATLAB, you can use the MATLAB avifile and addframe functions to create an AVI file. AVI files can be created using indexed and RGB images of classes uint8 and double, and don't require a multiframe image. For instructions on creating an AVI file, see the Development Environment section in the MATLAB documentation.

Displaying Multiple Images

MATLAB does not place any restrictions on the number of images you can display simultaneously. However, there are usually system limitations that are dependent on the computer hardware you are using. The sections below describe how to display multiple figures separately, or within the same figure.

The main limitation is the number of colors your system can display. This number depends primarily on the number of bits that are used to store the color information for each pixel. Most systems use either 8, 16, or 24 bits per pixel.

If you are using a system with 16 or 24 bits per pixel, you are unlikely to run into any problems, regardless of the number of images you display. However, if your system uses 8 bits per pixel, it can only display a maximum of 256 different colors, and you can therefore quickly run out of color slots if you display multiple images. (Actually, the total number of colors you can display is slightly fewer than 256, because some color slots are reserved for Handle Graphics objects. The operating system usually reserves a few colors as well.)

To determine the number of bits per pixel on your system, enter this command.

```
get(0,'ScreenDepth')
```

See "Working with Different Screen Bit Depths" on page 13-3 for more information.

This section discusses:

- Displaying each image in a separate figure
- Displaying multiple images in the same figure

It also includes information about working around system limitations.

Displaying Each Image in a Separate Figure

The simplest way to display multiple images is to display them in different figure windows. `imshow` always displays an image in the current figure, so if you display two images in succession, the second image replaces the first image. To avoid replacing the image in the current figure, use the `figure` command to explicitly create a new empty figure before calling `imshow` for the next image. For example,

```
imshow(I)
figure, imshow(I2)
figure, imshow(I3)
```

When you use this approach, the figures you create are empty initially.

If you have an 8-bit display, you must make sure that the total number of colormap entries does not exceed 256. For example, if you try to display three images, each having a different colormap with 128 entries, at least one of the images will display with the wrong colors. (If all three images have identical colormaps, there will not be a problem, because only 128 color slots are used.) Remember that intensity images are also displayed using colormaps, so the color slots used by these images count toward the 256-color total.

In the next example, two indexed images are displayed on an 8-bit display. Since these images do not have similar colormaps and due to the limitation of the screen color resolution, the first image is forced to use the colormap of the second image, resulting in an inaccurate display.

```
[X1,map1]=imread('forest.tif');
[X2,map2]=imread('trees.tif');
imshow(X1,map1),figure,imshow(X2,map2);
```



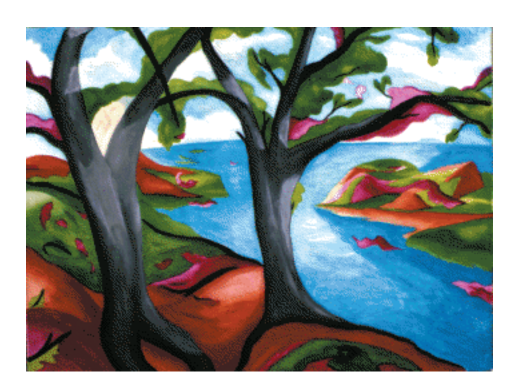


Figure 3-2: Displaying Two Indexed Images on an 8-bit Screen

As `X2` is displayed, `X1` is forced to use `X2`'s colormap (and now you can't see the forest for the trees). Note that the actual display results of this example will

vary depending on what other application windows are open and using up system color slots.

One way to avoid these display problems is to manipulate the colormaps to use fewer colors. There are various ways to do this, such as using the `imapprox` function. See "Reducing the Number of Colors in an Image" on page 13-6 for information.

Another solution is to convert images to RGB (truecolor) format for display, because MATLAB automatically uses dithering and color approximation to display these images. Use the `ind2rgb` function to convert indexed images to RGB.

```
imshow(ind2rgb(X,map))
```

Or, simply use the `cat` command to display an intensity image as an RGB image.

```
imshow(cat(3,I,I,I))
```

Displaying Multiple Images in the Same Figure

You can display multiple images in a single figure window with some limitations. This discussion shows you how to do this in one of two ways:

1. By using `imshow` in conjunction with `subplot`
2. By using `subimage` in conjunction with `subplot`

`subplot` divides a figure into multiple display regions. The syntax of `subplot` is

```
subplot(m,n,p)
```

This syntax divides the figure into an m-by-n matrix of display regions and makes the pth display region active.

For example, if you want to display two images side by side, use

```
[X1,map1]=imread('forest.tif');
[X2,map2]=imread('trees.tif');
subplot(1,2,1), imshow(X1,map2)
subplot(1,2,2), imshow(X2,map2)
```

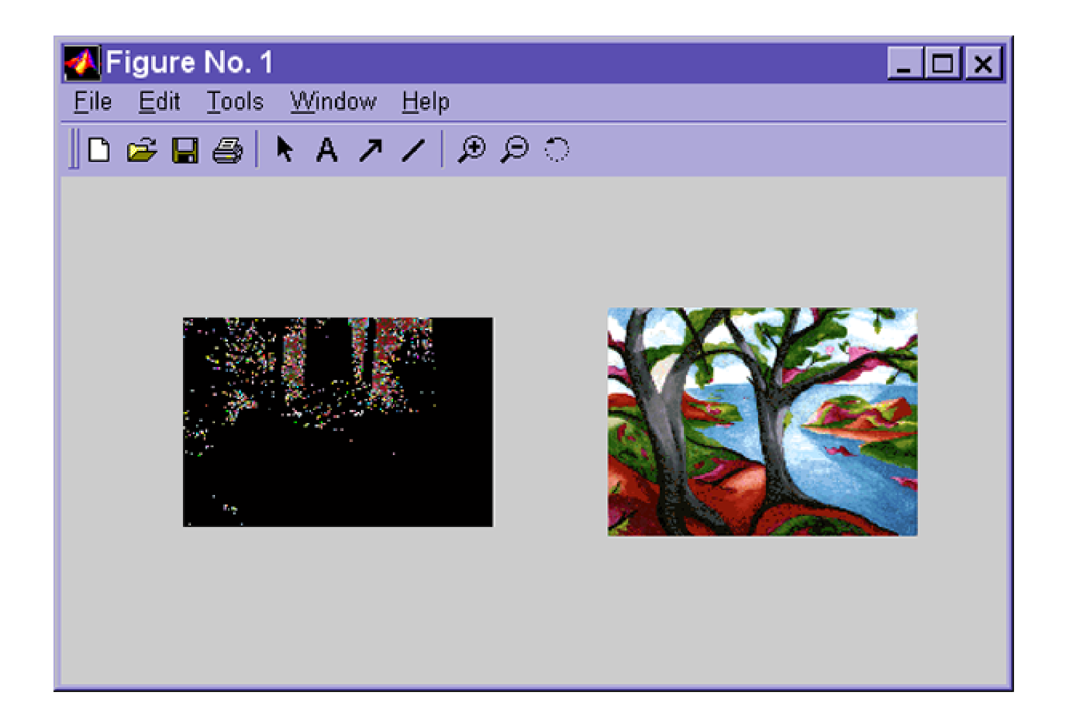


Figure 3-3: Two Images in Same Figure Using the Same Colormap

If sharing a colormap (using the `subplot` function) produces unacceptable display results as Figure 3-3 shows, use the `subimage` function (shown below). Or, as another alternative, you can map all images to the same colormap as you load them. See "Colormap Mapping" on page 13-11 for more information.

`subimage` converts images to RGB before displaying and therefore circumvents the colormap sharing problem. This example displays the same two images shown in Figure 3-3 with better results.

```
[X1,map1]=imread('forest.tif');
[X2,map2]=imread('trees.tif');
subplot(1,2,1), subimage(X1,map1)
subplot(1,2,2), subimage(X2,map2)
```

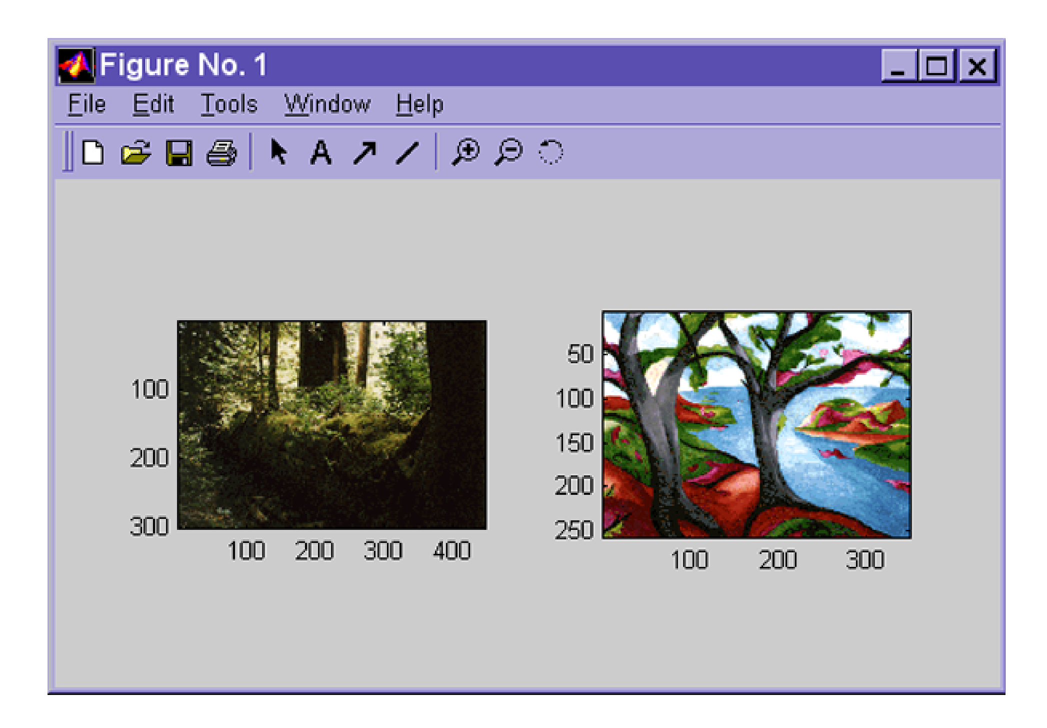


Figure 3-4: Two Images in Same Figure Using Separate Colormaps

Setting Toolbox Display Preferences

The behavior of imshow is influenced in part by the current settings of the *toolbox preferences*. Depending on the arguments you specify and the current settings of the toolbox preferences, imshow may:

- Suppress the display of axes and tick marks.
- Include or omit a "border" around the image.
- Call the truesize function to display the image without interpolation.
- Set other figure and axes properties to tailor the display.

All of these settings can be changed by using the iptsetpref function.The truesize preference can also be changed by setting the display_option parameter of imshow. This section describes how to set the toolbox preferences and how to use the display_option parameter.

When you display an image using the imshow function, MATLAB also sets the Handle Graphics figure, axes, and image properties, which control the way image data is interpreted. These settings are optimized for each image type. The specific properties set are described under the following sections:

- "The Image and Axes Properties of an Indexed Image" on page 3-4
- "The Image and Axes Properties of an Intensity Image" on page 3-6
- "The Image and Axes Properties of a Binary Image" on page 3-9
- "The Image and Axes Properties of an RGB Image" on page 3-10

Toolbox Preferences

The toolbox preferences affect the behavior of imshow for the duration of the current MATLAB session. You can change these settings at any time by using the iptsetpref function. To preserve your preference settings from one session

to the next, make your settings in your `startup.m` file. These are the preferences that you may set.

Toolbox Preference	Description
`ImshowBorder`	Controls whether `imshow` displays the figure window as larger than the image (leaving a border between the image axes and the edges of the figure), or the same size as the image (leaving no border).
`ImshowAxesVisible`	Controls whether `imshow` displays images with the axes box and tick labels.
`ImshowTruesize`	Controls whether `imshow` calls the `truesize` function. This preference can be overridden for a single call to `imshow`; see "Using the truesize Function" below for more details
`TrueSizeWarning`	Controls whether you will receive a warning message if an image is too large for the screen.

This example call to `iptsetpref` resizes the figure window so that it fits tightly around displayed images.

```
iptsetpref('ImshowBorder', 'tight');
```

To determine the current value of a preference, use the `iptgetpref` function.

For more information about toolbox preferences and the values they accept, see the reference entries for `iptgetpref` and `iptsetpref`.

Using the truesize Function

The `truesize` function assigns a single screen pixel to each image pixel, e.g., a 200-by-300 image will be 200 screen pixels in height and 300 screen pixels in width. This is generally the preferred way to display an image. In most situations, when the toolbox is operating under default behavior, `imshow` calls the `truesize` command automatically before displaying an image.

In some cases, you may not want `imshow` to automatically call `truesize` (for example, if you are working with a small image). If you display an image

without calling `truesize`, the image displays at the default axis size. In such cases, MATLAB must use *interpolation* to determine the values for screen pixels that do not directly correspond to elements in the image matrix. (See “Interpolation” on page 4-3 for more information.)

You can affect whether MATLAB automatically calls `truesize` by using either of these methods:

- Set the preference for the current MATLAB session. This example sets the `ImshowTruesize` preference to `'manual'`, meaning that `truesize` will not be automatically called by `imshow`.

  ```
  iptsetpref('ImshowTruesize','manual')
  ```

- Set the preference for a single `imshow` command by setting the `display_option` parameter. This example sets the `display_option` parameter to `truesize`, so that `truesize` is called for the image displayed, regardless of the current preference setting.

  ```
  imshow(X, map,'truesize')
  ```

For more information see the reference descriptions for `imshow` and `truesize`.

Zooming in on a Region of an Image

The simplest way to zoom in on a region of an image is to use the zoom buttons provided on the figure window. To enable zooming from the command line, use the `zoom` command.When you zoom in, the figure window remains the same size, but only a portion of the image is displayed, at a higher magnification. (`zoom` works by changing the axis limits; it does not change the image data in the figure.)

Once zooming in is enabled, there are two ways to zoom in on an image:

1. Single mouse click: click on a spot in the image by placing the cursor on the spot and the pressing the left mouse button. The image is magnified and the center of the new view is the spot where you clicked.

2. Click and drag the mouse: select a region by clicking on the image, holding down the left mouse button, and dragging the mouse. This creates a dotted rectangle. When you release the mouse button, the region enclosed by the rectangle is displayed with magnification.

Zooming In or Out with the Zoom Buttons

The zoom buttons in the MATLAB figure enable you to zoom in or out on an image using your mouse.



To zoom in, click the “magnifying glass” button with the plus sign in it. There are two ways to zoom in on an image after selecting the zoom in button. See “Zooming in on a Region of an Image” above.



To zoom out, click the “magnifying glass” button with the minus sign in it. Click your left mouse button over the spot in the image you would like to zoom out from.

Zooming In or Out from the Command Line

The `zoom` command enables you to zoom in or out on an image using your mouse.

To enable zooming (in or out), type

```
zoom on
```

There are two ways to zoom in on an image. See "Zooming in on a Region of an Image" on page 3-26.

To zoom out, click on the image with the right mouse button. (If you have a single-button mouse, hold down the **Shift** key and click.)

To zoom out completely and restore the original view, enter

```
zoom out
```

To disable zooming, enter

```
zoom off
```

Texture Mapping

When you use the `imshow` command, MATLAB displays the image in a two-dimensional view. However, it is also possible to map an image onto a parametric surface, such as a sphere, or below a surface plot. The `warp` function creates these displays by *texture mapping* the image. Texture mapping is a process that maps an image onto a surface grid using interpolation.

This example texture maps an image of a test pattern onto a cylinder.

```
[x,y,z] = cylinder;
I = imread('testpat1.tif');
warp(x,y,z,I);
```

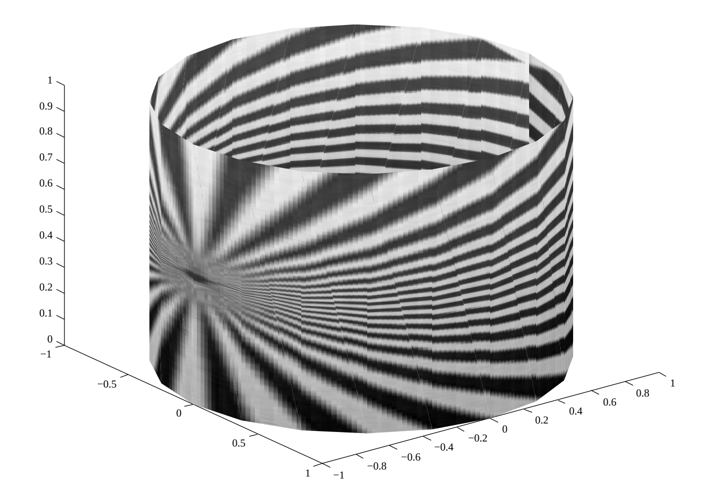


Figure 3-5: An Image Texture Mapped onto a Cylinder

The image may not map onto the surface in the way that you had expected. One way to modify the way the texture map appears is to change the settings of the `Xdir`, `Ydir`, and `Zdir` properties. For more information, see Changing Axis Direction in the MATLAB Graphics documentation.

For more information about texture mapping, see the reference entry for the `warp` function.

Printing Images

If you want to output a MATLAB image to use in another application (such as a word-processing program or graphics editor), use imwrite to create a file in the appropriate format. See “Writing a Graphics Image” on page 2-17 for details.

If you want to print the contents of a MATLAB figure (including nonimage elements such as labels), use the MATLAB print command, or choose the **Print** option from the **File** menu of the figure window. Note that if you produce output in either of these ways, the results reflect the settings of various Handle Graphics properties. In some cases, you may need to change the settings of certain properties to get the results you want.

Here are some tips that may be helpful when you print images:

- Image colors print as shown on the screen. This means that images are not affected by the InvertHardcopy figure property.
- To ensure that printed images have the proper size and aspect ratio, you should set the figure’s PaperPositionMode property to auto. When PaperPositionMode is set to auto, the width and height of the printed figure are determined by the figure’s dimensions on the screen. By default, the value of PaperPositionMode is manual. If you want the default value of PaperPositionMode to be auto, you can add this line to your startup.m file.

```
set(0,'DefaultFigurePaperPositionMode','auto')
```

For detailed information about printing with **File/Print** or the print command (and for information about Handle Graphics), see “Printing and Exporting Figures with MATLAB” in the MATLAB Graphics documentation. For a complete list of options for the print command, enter help print at the MATLAB command line prompt or see print in the MATLAB Function Reference.

Troubleshooting

This section contains three common scenarios (in bold text) which can occur unexpectedly, and what you can do to derive the expected results.

Color Image Displays as Grayscale

If a color image displays as a grayscale image, it can indicate that the image is an indexed image and you have not specified the associated colormap.

To display an indexed image using `imshow`, you must specify the colormap, using this syntax:

```
imshow(X,map);
```

You should also make sure that you used the correct syntax for loading an indexed image, which is,

```
[X, map]=imread('filename.ext');
```

For more information about displaying indexed images, see “Displaying Indexed Images” on page 3-3.

Displaying Multiframe Images

If you load a multiframe image and display it, MATLAB only displays one frame.

To view the other frames in a multiframe image, you must load each frame separately. This can be done using a `for` loop. It may be helpful to first use `imfinfo` to find out how many frames the image file contains, and what their dimensions are. To see an example, go to “Displaying the Frames of a Multiframe Image Individually” on page 3-14.

Spatial Transformations

This section describes the spatial transformation functions in the Image Processing Toolbox. Spatial transformations map pixel locations in an input image to new locations in an output image. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Aliasing	Artifacts in an image that can appear as a result of reducing an image's size. When the size of an image is reduced, original pixels are downsampled to create fewer pixels. Aliasing that occurs as a result of size reduction normally appears as "stair-step" patterns (especially in high contrast images), or as "Moire" (ripple-effect) patterns.
Anti-aliasing	Any method for correcting aliasing (see above). The method discussed in this chapter is low-pass filtering (see below).
Bicubic interpolation	Output pixel values are calculated from a weighted average of pixels in the nearest 4-by-4 neighborhood.
Bilinear interpolation	Output pixel values are calculated from a weighted average of pixels in the nearest 2-by-2 neighborhood.
Geometric operation	An operation that modifies the spatial relations between pixels in an image. Examples include resizing (growing or shrinking), rotating, and shearing.
Interpolation	The process by which we estimate an image value at a location in between image pixels.
Nearest neighbor interpolation	Output pixel values are assigned the value of the pixel that the point falls within. No other pixels are considered.

Interpolation

Interpolation is the process by which we estimate an image value at a location in between image pixels. For example, if you resize an image so it contains more pixels than it did originally, the software obtains values for the additional pixels through interpolation. The `imresize` and `imrotate` geometric functions use two-dimensional interpolation as part of the operations they perform. (The `improfile` image analysis function also uses interpolation. See "Intensity Profile" on page 10-4 for information about this function.)

The Image Processing Toolbox provides three interpolation methods:

- Nearest neighbor interpolation
- Bilinear interpolation
- Bicubic interpolation

The interpolation methods all work in a fundamentally similar way. In each case, to determine the value for an interpolated pixel, you find the point in the input image that the output pixel corresponds to. You then assign a value to the output pixel by computing a weighted average of some set of pixels in the vicinity of the point. The weightings are based on the distance each pixel is from the point.

The methods differ in the set of pixels that are considered:

- For nearest neighbor interpolation, the output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered.
- For bilinear interpolation, the output pixel value is a weighted average of pixels in the nearest 2-by-2 neighborhood.
- For bicubic interpolation, the output pixel value is a weighted average of pixels in the nearest 4-by-4 neighborhood.

The number of pixels considered affects the complexity of the computation. Therefore the bilinear method takes longer than nearest neighbor interpolation, and the bicubic method takes longer than bilinear. However, the greater the number of pixels considered, the more accurate the effect is, so there is a trade-off between processing time and quality.

Image Types

The functions that use interpolation take an argument that specifies the interpolation method. For most of these functions, the default method is nearest neighbor interpolation. This method produces acceptable results for all image types, and is the only method that is appropriate for indexed images. For intensity and RGB images, however, you should generally specify bilinear or bicubic interpolation, because these methods produce better results than nearest neighbor interpolation.

For RGB images, interpolation is performed on the red, green, and blue image planes individually.

For binary images, interpolation has effects that you should be aware of. If you use bilinear or bicubic interpolation, the computed values for the pixels in the output image will not all be 0 or 1. The effect on the resulting output image depends on the class of the input image:

- If the class of the input image is `double`, the output image is a grayscale image of class `double`. The output image is not binary, because it includes values other than 0 and 1.
- If the class of the input image is `uint8`, the output image is a binary image of class `uint8`. The interpolated pixel values are rounded off to 0 and 1 so the output image can be of class `uint8`.

Note For bicubic interpolation, you may need to damp doubles to within the [0 1] range.

If you use nearest neighbor interpolation, the result is always binary, because the values of the interpolated pixels are taken directly from pixels in the input image.

Image Resizing

To change the size of an image, use the `imresize` function. Using `imresize`, you can:

- Specify the size of the output image.
- Specify the interpolation method used
- Specify the filter to use to prevent aliasing

Specifying the Size of the Output Image

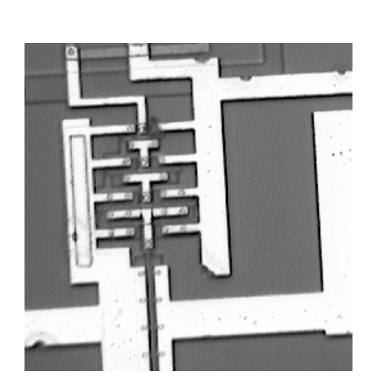
Using `imresize`, you can specify the size of the output image in two ways

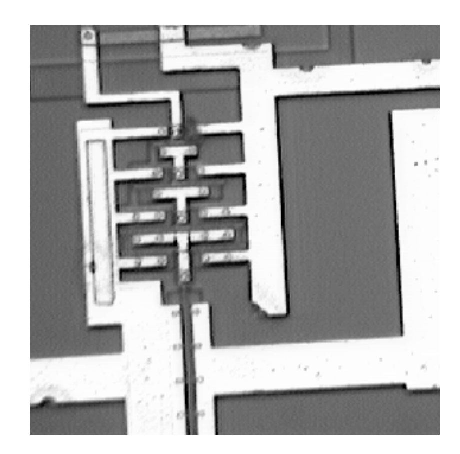
- By specifying the magnification factor to be used on the image
- By specifying the dimensions of the output image

Using the Magnification Factor

To enlarge an image, specify a magnification factor greater than 1. To reduce an image, specify a magnification factor between 0 and 1. For example, the command below increases the size of the image, `I`, by 1.25 times.

```
I = imread('ic.tif');
J = imresize(I,1.25);
imshow(I)
figure, imshow(J)
```





Specifying the Size of the Output Image

You can specify the size of the output image by passing a vector that contains the number of row and columns in the output image. The command below creates an output image, Y, with 100 rows and 150 columns.

```
Y = imresize(X,[100 150])
```

Note If the specified size does not produce the same aspect ratio as the input image, the output image will be distorted.

Specifying the Interpolation Method

By default, `imresize` uses nearest neighbor interpolation to determine the values of pixels in the output image but you can specify other interpolation methods. This table lists the supported interpolation methods. See "Interpolation" for more information about these methods.

Argument Value	Interpolation Method
'nearest'	Nearest neighbor (the default)
'bilinear'	Bilinear interpolation
'bicubic'	Bicubic interpolation

In this example, `imresize` uses the bilinear interpolation method.

```
Y = imresize(X,[100 150],'bilinear')
```

Using Filters to Prevent Aliasing

Reducing the size of an image can introduce artifacts, such as aliasing, in the output image because information is always lost when you reduce the size of an image. Aliasing appears as ripple patterns (called Moire patterns) in the output image.

When you reduce the size of the image using either bilinear or bicubic interpolation, `imresize` automatically applies a low-pass filter to the image

before interpolation, to limit the impact of aliasing on the output image. You can specify the size of this filter or specify a different filter.

Note Even with low-pass filtering, resizing can introduce artifacts, because information is always lost when you reduce the size of an image.

The `imresize` function does not apply a low-pass filter if nearest neighbor interpolation is used. Nearest neighbor interpolation is primarily used for indexed images, and low-pass filtering is not appropriate for these images.

You can also specify a filter of your own creation. For more information about specifying a filter, see the reference page for `imresize`.

Image Rotation

To rotate an image, use the imrotate function. imrotate accepts two primary arguments:

- The image to be rotated
- The rotation angle

You specify the rotation angle in degrees. If you specify a positive value, imrotate rotates the image counterclockwise; if you specify a negative value, imrotate rotates the image clockwise. This example rotates the image, I, 35 degrees in the counterclockwise direction.

```
J = imrotate(I,35);
```

As optional arguments to imrotate, you can also specify:

- The interpolation method
- The size of the output image

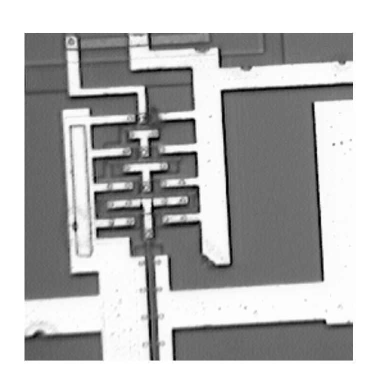
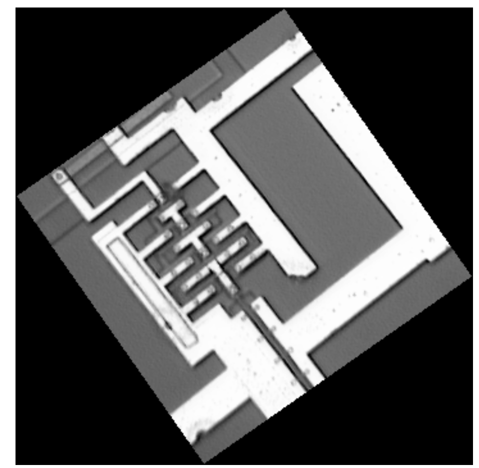
Specifying the Interpolation Method

By default, imrotate uses nearest neighbor interpolation to determine the value of pixels in the output image but you can specify other interpolation methods. This table lists the supported interpolation methods. See "Interpolation" for more information about these methods.

Argument Value	Interpolation Method
'nearest'	Nearest neighbor (the default)
'bilinear'	Bilinear interpolation
'bicubic'	Bicubic interpolation

For example, these commands rotate an image 35° counterclockwise and use bilinear interpolation.

```
I = imread('ic.tif');
J = imrotate(I,35,'bilinear');
imshow(I)
figure, imshow(J)
```

Specifying the Size of the Output Image

By default, `imrotate` creates an output image large enough to include the entire original image. Pixels that fall outside the boundaries of the original image are set to 0 and appear as a black background in the output image. If you specify the text string `crop'` as an argument, `imrotate` crops the output image to be the same size as the input image. (See the reference page for `imrotate` for an example of cropping.)

Image Cropping

To extract a rectangular portion of an image, use the imcrop function. imcrop accepts two primary arguments:

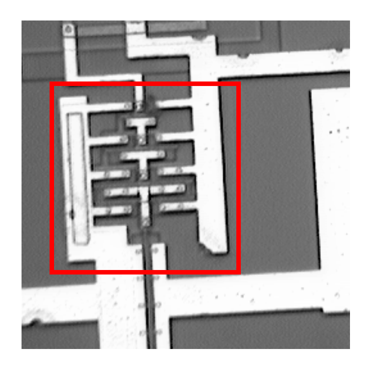
- The image to be cropped
- The coordinates of a rectangle that defines the crop area

If you call imcrop without specifying the crop rectangle, the cursor changes to a cross hair when it is over the image. Click on one corner of the region you want to select, and while holding down the mouse button, drag across the image. imcrop draws a rectangle around the area you are selecting. When you release the mouse button, imcrop creates a new image from the selected region.

In this example, you display an image and call imcrop. The rectangle you select is shown in red.

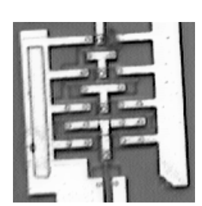
```
imshow ic.tif
I = imcrop;
```

The imcrop function waits for you to draw the cropping rectangle on the image.



Now call imshow to display the cropped image. If you call imcrop without specifying any output arguments, imcrop displays the image in a new figure.

```
imshow(I)
```



Performing General Spatial Transformations

To perform general 2-D spatial transformations, use the `imtransform` function. (For information about performing more advanced transformations, see “Advanced Spatial Transformation Techniques” on page 4-15.)

The `imtransform` function accepts two primary arguments:

- The image to be transformed
- A spatial transformation structure, called a `TFORM`, that specifies the type of transformation you want to perform

Specifying the Transformation Type

You specify the type of transformation you want to perform in a `TFORM` structure. There are two ways to create a `TFORM` struct:

- Using the `maketform` function
- Using the `cp2tform` function

Using maketform

When you use the `maketform` function, you can specify the type of transformation you want to perform. The following table lists the types of transformations `maketform` supports.

Transformation	Description
`'affine'`	A transformation that may include translation, rotation, scaling, stretching and shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles may become parallelograms.
`'projective'`	A transformation in which straight lines remain straight, but parallel lines converge toward “vanishing points.” (The vanishing points may fall inside or outside the image—even at infinity.)
`'box'`	A special case of an affine transformation where each dimension is shifted and scaled independently.

Transformation	Description
`'custom'`	A user-defined transformation, providing the forward and/or inverse functions that are called by `imtransform`.
`'composite'`	A composition of two or more transformations.

The `'custom'` and `'composite'` capabilities of `maketform` allow a virtually limitless variety of spatial transformations to be used with `imtransform` and/or `tformarray`.

Using cp2tform

You use `cp2tform` to create the `TFORM` when you want to perform a transformation that requires fitting of data points, such as a polynomial transformation. Chapter 5, "Image Registration" explains how to use the `cp2tform` function to fit a 2-D transformation to a set of control points selected in a pair of images.

Note When used with `imtransform`, `TFORM` structures must define a 2-D spatial transformation. If an image contains more than two dimensions, such as an RGB image, the same 2-D transformation is automatically applied to all 2-D planes along the higher dimensions. For arbitrary-dimensional array transformations, use the `tformarray` function.

Performing the Transformation

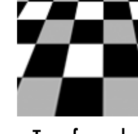
Once you define the transformation in a TFORM struct, you can perform the transformation by calling imtransform.

For example, this code uses imtransform to perform a projective transformation of a checkerboard image.

```
I = checkerboard(20,1,1);
figure; imshow(I)
T = maketform('projective',[1 1; 41 1; 41 41;   1 41],...
              [5 5; 40 5; 35 30; -10 30]);
R = makeresampler('cubic','circular');
K = imtransform(I,T,R,'Size',[100 100],'XYScale',1);
figure, imshow(K)
```



Original image



Transformed image

The imtransform function options let you control many aspects of the transformation. For example, note how the transformed image appears to contain multiple copies of the original image. This is accomplished by using the 'Size' option, to make the output image larger than the input image, and then specifying a padding method that extends the input image by repeating the pixels in a circular pattern. The Image Processing Toolbox Image Transformation demos provide more examples of using the imtransform function, and related functions, to perform different types of spatial transformations.

Advanced Spatial Transformation Techniques

The following functions, when used in combination, provide a vast array of options for defining and working with 2-D, N-D, and mixed-D spatial transformations:

- `maketform`
- `fliptform`
- `tformfwd`
- `tforminv`
- `findbounds`
- `makeresampler`
- `tformarray`
- `imtransform`

The `imtransform`, `findbounds`, and `tformarray` functions use the `tformfwd` and `tforminv` functions internally to encapsulate the forward transformations needed to determine the extent of an output image or array and/or to map the output pixels/array locations back to input locations. You can use `tformfwd` and `tforminv` to explore the geometric effects of a transformation by applying them to points and lines and plotting the results. They support a consistent handling of both image and pointwise data.

The previous example, "Performing the Transformation" on page 4-14, used the `makeresampler` function with a standard interpolation method. You can also use it to obtain special effects or custom processing. For example, you could specify your own separable filtering/interpolation kernel, build a custom resampler around the MATLAB `interp2` or `interp3` functions, or even implement an advanced anti-aliasing technique.

And, as noted, you can use `tformarray` to work with arbitrary-dimensional array transformations. The arrays do not even need to have the same dimensions. The output can either have a lower or higher number of dimensions than the input.

For example, if you are sampling 3-D data on a 2-D slice or manifold, the input array might have a lower dimensionality. The output dimensionality might be higher, for example, if you combine multiple 2-D transformations into a single 2-D to 3-D operation.

Image Registration

This section describes the image registration capabilities of the Image Processing Toolbox. Image registration is the process of aligning two or more images of the same scene. Image registration is often used as a preliminary step in other image processing applications. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	MATLAB Definition
Aligned image	The output image after registration has been performed. The output image is derived by applying a transformation to the input image (see below) that brings it into alignment with the base image (see below).
Base image	This is the image against which you compare the image to be registered. It is also often called the *reference* image.
Control point pairs	Matching locations, also referred to as *landmarks*, in the input image and the base image.
Distortion	The differences in one image as compared to another of the same subject. These may have occurred as a result of terrain relief and other changes in perspective when imaging the same scene from different viewpoints, lens and other internal sensor distortions, or differences between sensors and sensor types.
Global transformation	A transformation in which a single mathematical expression applies to an entire image.
Input image	This refers to the image that you wish to register. It is often called the *observed* image.

Terms	MATLAB Definition
Local transformation	A transformation in which different mathematical expressions (usually differing in parameters rather than form) apply to different regions within an image.
Spatial transformation	The mapping of locations of points in one image to new locations in another image.

Registering an Image

Image registration is the process of aligning two or more images of the same scene. Typically, one image, called the base image, is considered the reference to which the other images, called input images, are compared. The object of image registration is to bring the input image into alignment with the base image by applying a spatial transformation to the input image.

A spatial transformation maps locations in one image to new locations in another image. (For more details, see Chapter 4, "Spatial Transformations.") Determining the parameters of the spatial transformation needed to bring the images into alignment is key to the image registration process.

Image registration is often used as a preliminary step in other image processing applications. For example, you can use image registration to align satellite images of the earth's surface or images created by different medical diagnostic modalities (MRI and SPECT). After registration, you can compare features in the images to see how a river has migrated, how an area is flooded, or to see if a tumor is visible in an MRI or SPECT image.

Point Mapping

The Image Processing Toolbox provides tools to support point mapping to determine the parameters of the transformation required to bring an image into alignment with another image. In point mapping, you pick points in a pair of images that identify the same feature or landmark in the images. Then, a spatial mapping is inferred from the positions of these control points.

Image registration using point mapping involves these steps:

1. Read the images into the MATLAB workspace.
2. Specify control point pairs in the images.
3. Save the control point pairs.
4. Fine tune the control points using cross-correlation. (This is an optional step.)
5. Specify the type of transformation to be used and infer its parameters from the control point pairs.

6 Transform the unregistered image to bring it into alignment.

This process is best understood by looking at an example. See "Example: Registering to a Digital Orthophoto" on page 5-6 for an extended example.

Note You may need to perform several iterations of this process, experimenting with different types of transformations, before you achieve a satisfactory result. In some cases, you may perform successive registrations, removing gross global distortions first, and then removing smaller local distortions in subsequent passes.

The following figure, Overview of Image Registration Process, provides a graphic illustration of this process.

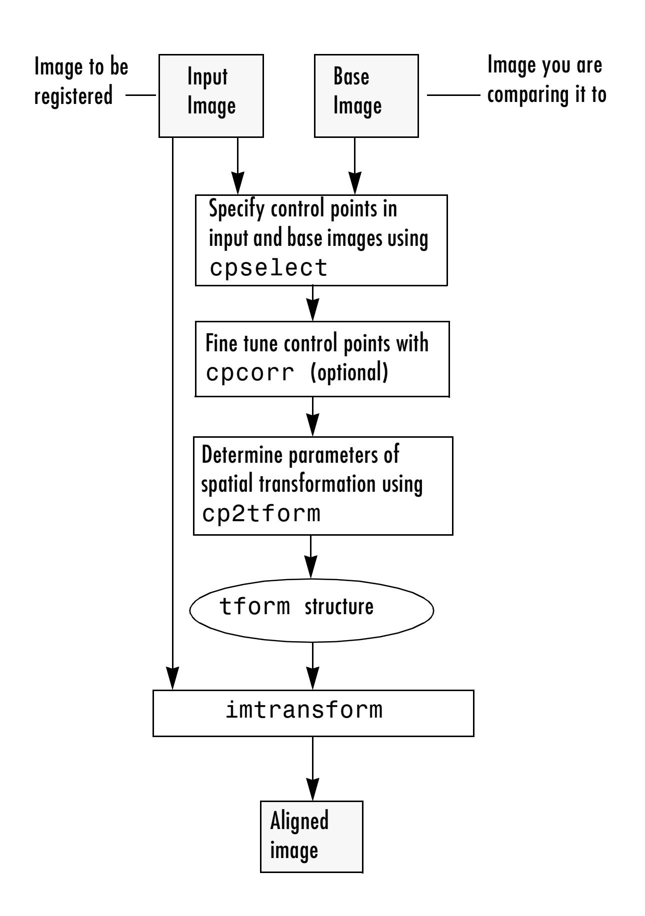


Figure 5-1: Overview of Image Registration Process

Example: Registering to a Digital Orthophoto

This example registers a digital aerial photograph to a digital orthophoto covering the same area. Both images are centered on the business district of West Concord, Massachusetts.

The aerial image is geometrically uncorrected: it includes camera perspective, terrain and building relief, and internal (lens) distortions, and it does not have any particular alignment or registration with respect to the earth.

The orthophoto, supplied by the Massachusetts Geographic Information System (MassGIS), has been orthorectified to remove camera, perspective, and relief distortions (via a specialized image transformation process). It is also georegistered (and geocoded)—the columns and rows of the digital orthophoto image are aligned to the axes of the Massachusetts State Plane coordinate system, each pixel center corresponds to a definite geographic location, and every pixel is 1 meter square in map units.

Step 1: Read the Images into MATLAB

In this example, the base image is `westconcordorthophoto.png`, the MassGIS georegistered orthophoto. It is a panchromatic (grayscale) image. The image to be registered is `westconcordaerial.png`, a digital aerial photograph supplied by mPower3/Emerge, and is a visible-color RGB image.

```
orthophoto = imread('westconcordorthophoto.png');
figure, imshow(orthophoto)
unregistered = imread('westconcordaerial.png');
figure, imshow(unregistered)
```

You do not have to read the images into the MATLAB workspace. The `cpselect` function accepts file specifications for grayscale images. However, if you want to use cross-correlation to tune your control point positioning, the images must be in the workspace.



Aerial Photo Image



Orthophoto Image

Step 2: Choose Control Points in the Images

The toolbox provides an interactive tool, called the Control Point Selection Tool, that you can use to pick pairs of corresponding control points in both images. Control points are landmarks that you can find in both images, like a road intersection, or a natural feature.

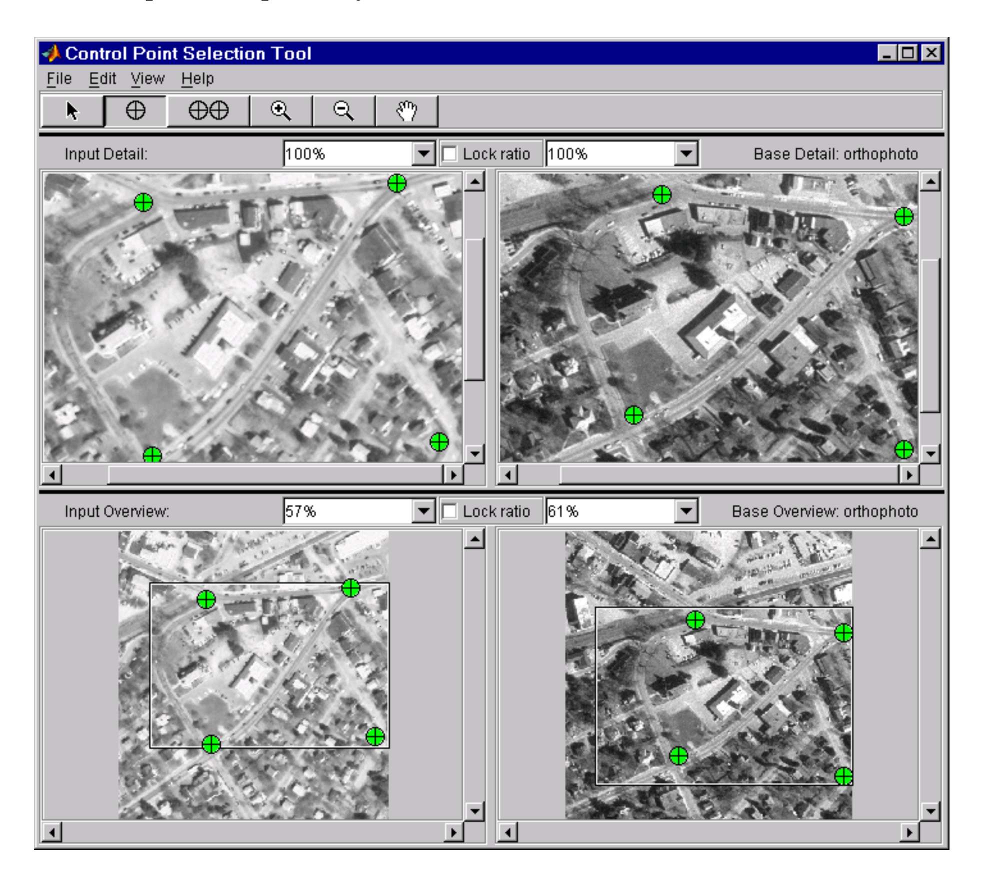
To start this tool, enter `cpselect` at the MATLAB prompt, specifying as arguments the input and base images.

Note The unregistered image is an RGB image. Because the Control Point Selection Tool only accepts grayscale images, the example passes only one plane of the color image to `cpselect`.

```
cpselect(unregistered(:,:,1),orthophoto)
```

The `cpselect` function displays two views of both the input image and the base image in which you can pick control points by pointing and clicking. For more

information, see “Selecting Control Points” on page 5-15. This figure shows the Control Point Selection Tool with four pairs of control points selected. The number of control point pairs you pick is at least partially determined by the type of transformation you want to perform (specified in Step 5). See “Types of Supported Transformations” on page 5-13 for information about the minimum number of points required by each transformation.



Step 3: Save the Control Point Pairs to the MATLAB Workspace

In the Control Point Selection Tool, click on the **File** menu and choose the **Save Points to Workspace** option. See “Saving Control Points” on page 5-30 for more information.

For example, the Control Point Selection Tool returns the following set of control points in the input image. These values represent spatial coordinates; the left column are x coordinates, the right column are y coordinates.

```
input_points =
  120.7086   93.9772
  319.2222   78.9202
  127.9838  291.6312
  352.0729  281.1445
```

Step 4: Fine-tune the Control Point Pair Placement

This is an optional step that uses cross-correlation to adjust the position of the control points you selected with `cpselect`. See "Using Correlation to Improve Control Points" on page 5-33 for more information.

Note `cpcorr` can only adjust points for images that are the same scale and have the same orientation. The images cannot be rotated relative to each other. Because the Concord image is rotated in relation to the base image, `cpcorr` cannot tune the control points. When it cannot tune the points, `cpcorr` returns the input points, unmodified.

```
input_points_corr = cpcorr(input_points,base_points,...
                           unregistered(:,:,1),orthophoto)
input_points_corr =
  120.7086   93.9772
  319.2222   78.9202
  127.1046  289.8935
  352.0729  281.1445
```

Step 5: Specify the Type of Transformation and Infer its Parameters

In this step, you pass the control points to the `cp2tform` function that determines the parameters of the transformation needed to bring the image into alignment. `cp2tform` is a data-fitting function that determines the transformation based on the geometric relationship of the control points. `cp2tform` returns the parameters in a geometric transformation structure, called a `TFORM` structure.

When you use cp2tform, you must specify the type of transformation you want to perform. The cp2tform function can infer the parameters for five types of transformations. You must choose which transformation will correct the type of distortion present in the input image. See "Types of Supported Transformations" on page 5-13 for more information. Images can contain more than one type of distortion.

The predominant distortion in the aerial image of West Concord (the input image) results from the camera perspective. Ignoring terrain relief, which is minor in this area, image registration can correct for this using a projective transformation. The projective transformation also rotates the image into alignment with the map coordinate system underlying the base image (the digital orthophoto). Other distortions could be corrected simultaneously by first creating a composite transformation with maketform, given sufficient information about the terrain and camera. (See "Performing General Spatial Transformations" on page 4-12 for more information.)

```
mytform = cp2tform(input_points,base_points,'projective');
```

Step 6: Transform the Unregistered Image

As the final step in image registration, you transform the input image to bring it into alignment with the base image. You use imtransform to perform the transformation, passing it the input image and the TFORM structure, which defines the transformation. imtransform returns the transformed image. For more information about using imtransform, see Chapter 4, "Spatial Transformations."

```
registered = imtransform(unregistered,mytform)
```

Note The transformation defined in mytform, which is based on control points picked in only one plane of the RGB image, is applied to all three planes of the input image.

Compare the transformed image to the base image to see how the registration came out.



Registered Image



Orthophoto Image

Types of Supported Transformations

The `cp2tform` function can infer the parameters for six types of transformations. This table lists the transformations in order of complexity, with examples of each type of distortion.

The first four transformations, `'linear conformal'`, `'affine'`, `'projective'` and `'polynomial'` are global transformations. In these transformations, a single mathematical expression applies to an entire image. The last two transformations, `'piecewise linear'` and `'lwm'` (local weighted mean), are local transformations. In these transformations, different mathematical expressions apply to different regions within an image.

When exploring how different transformations affect the images you are working with, try the global transformations first. If these transformation are not satisfactory, try the local transformations; the piecewise linear transformation first and then the local weighted mean transformation.

Transformation Type	Description	Minimum Control Points	Example
`'linear conformal'`	Use this transformation when shapes in the input image are unchanged, but the image is distorted by some combination of translation, rotation, and scaling. Straight lines remain straight, and parallel lines are still parallel.	2 pairs	
`'affine'`	Use this transformation when shapes in the input image exhibit shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles become parallelograms.	3 pairs	

'projective'	Use this transformation when the scene appears "tilted." Straight lines remain straight, but parallel lines converge toward "vanishing points" (which may or may not fall within the image.	4 pairs	
'polynomial'	Use this transformation when objects in the image are curved. The higher the order of the polynomial, the better the fit, but the result can contain more curves than the base image.	6 pairs (order 2) 10 pairs (order 3) 16 pairs (order 4)	
'piecewise linear'	Use this transformation when parts of the image appear distorted differently.	4 pairs	
'lwm'	Use this transformation (local weighted mean), when the distortion varies locally and piecewise linear is not sufficient.	6 pairs (12 pairs recommended)	

Selecting Control Points

The toolbox includes an interactive tool that enables you to specify control points in the images you want to register. The tool displays the images side-by-side. When you are satisfied with the number and placement of the control points, you can save the control points.

Using the Control Point Selection Tool

To specify control points in a pair of images you want to register, use the Control Point Selection Tool, `cpselect`. The tool displays the image you want to register, called the *input* image, next to the image you want to compare it to, called the *base* image or *reference* image.

Specifying control points is a four-step process:

1. Start the tool, specifying the input image and the base image.
2. View the images, looking for visual elements that you can identify in both images. `cpselect` provides many ways to navigate around the image, panning and zooming to view areas of the image in more detail.
3. Specify matching control point pairs in the input image and the base image.
4. Save the control points in the MATLAB workspace.

Figure 5-2, Control Point Selection Tool shows the default appearance of the tool when you first start it. To get more information about any part of the interface, click on it in this figure.

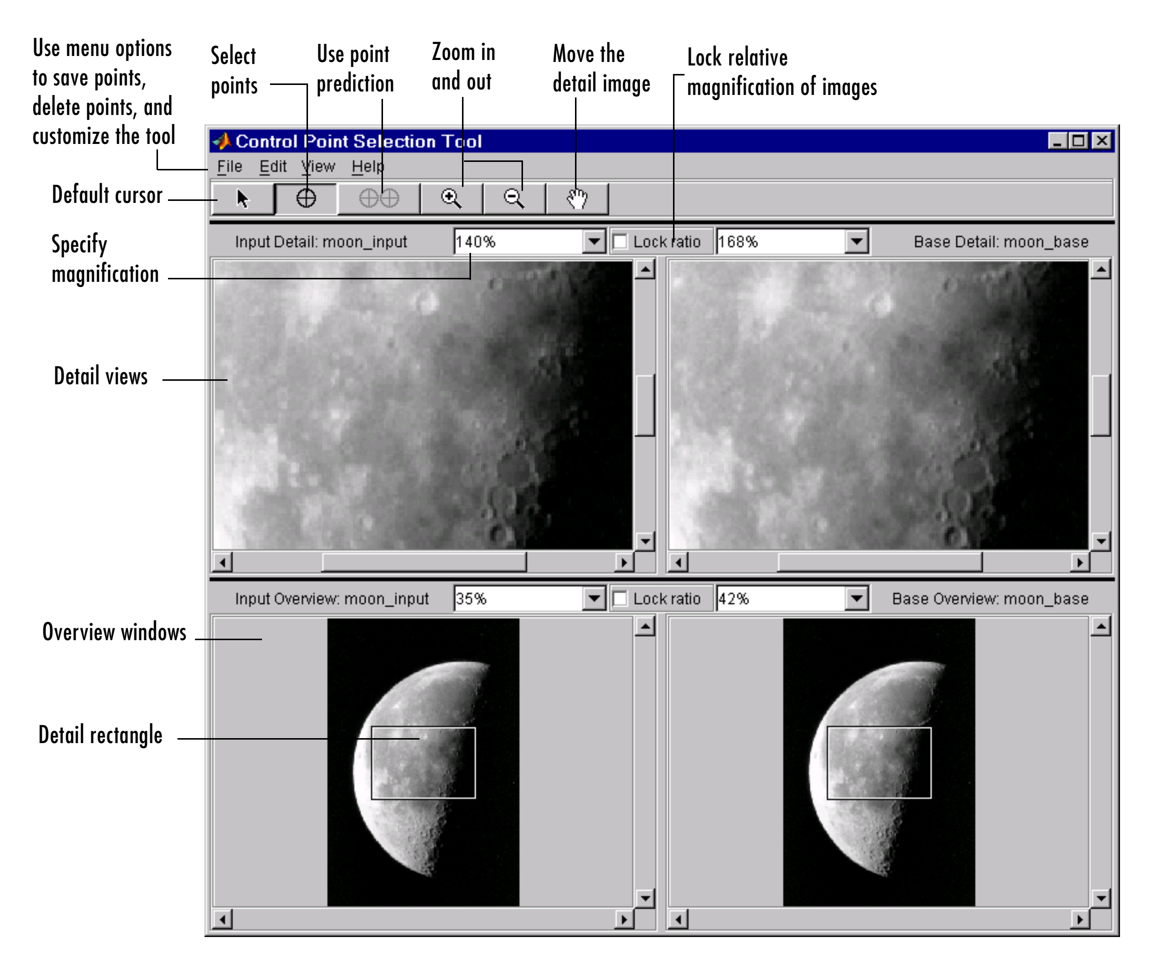


Figure 5-2: Control Point Selection Tool

Starting the Control Point Selection Tool

To use the Control Point Selection Tool, enter the `cpselect` command at the MATLAB prompt. As arguments, specify the image you want to register (the input image), and the image you want to compare it to (the base image).

To illustrate, this code fragment reads an image into a variable, `moon_base`, in the MATLAB workspace. It then creates another version of the image with a

deliberate size distortion, called `moon_input`. This is the image that needs registration to remove the size distortion. The code then starts the `cpselect` tool, specifying the two images.

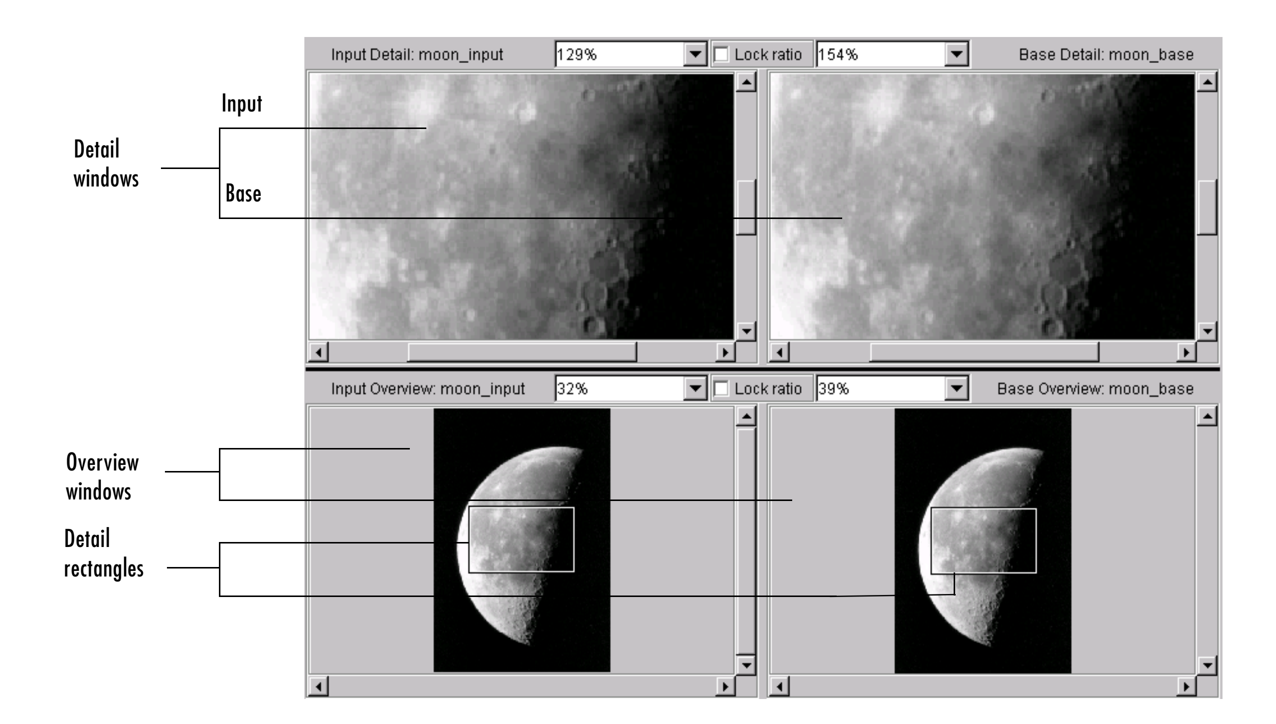
```
moon_base = imread('moon.tif );
moon_input = imresize(moon_base, 1.2);
cpselect(moon_input, moon_base);
```

The `cpselect` command has other optional arguments. For example, you can restart a control point selection session by including a `cpstruct` structure as the third argument. For more information about restarting sessions, see "Saving Control Points" on page 5-30. For complete details, see the `cpselect` reference page.

Default Views of the Images

When the Control Point Selection Tool starts, it contains four image display windows. The top two windows are called the **Detail** windows. These windows show a closeup view of a portion of the images you are working with. The input image is on the left and the base image is on the right. The two windows at the bottom of the interface are called the **Overview** windows. These windows show the images in their entirety, at the largest scale that fits the window. The input overview image is on the left and the base overview image is on the right.

Superimposed on the image in the **Overview** windows is a rectangle, called the **Detail Rectangle**. This rectangle defines the part of the image that is visible in the **Detail** window. By default, at startup, the **Detail Rectangle** covers one quarter of the entire image and is positioned over the center of the image.



Viewing the Images

By default, `cpselect` displays the entire base and input images in the **Overview** windows and displays a close up view of a portion of these images in the **Detail** windows. However, to find visual elements that are common to both images, you may want to change the section of the image displayed in the detail view or zoom in on a part of the image to view it in more detail. The following sections describe the different ways to change your view of the images.

- "Locking the Relative Magnification of the Input and Base Images" on page 5-22

Using Scroll Bars to View Other Parts of an Image

To view parts of an image that are not visible in the **Detail** or **Overview** windows, use the scroll bars provided in each window.

As you scroll the image in the **Detail** window, note how the **Detail Rectangle** moves over the image in the **Overview** window. The position of the **Detail Rectangle** always shows the portion of the image in the **Detail** window.

Using the Detail Rectangle to Change the View

To get a closer view of any part of the image, move the **Detail Rectangle** in the **Overview** window over that section of the image. `cpselect` displays that section of the image in the **Detail** window at a higher magnification than the overview window.

To move the **Detail Rectangle**:

1. Click on the default cursor button in the button bar.
2. Move the pointer into the **Detail Rectangle**. The cursor changes to the fleur shape, ,indicating the directions in which it can be moved.
3. Press and hold the mouse button to drag the **Detail Rectangle** anywhere on the image.

Note As you move the **Detail Rectangle** over the image in the **Overview** window, the view of the image displayed in the **Detail** window changes.

Moving the Image Displayed in the Detail Window

To change the section of the image displayed in the **Detail** window, use the Hand tool to move the image in the window.

To use the Hand tool:

1. Click on the Hand tool button in the button bar.

2 Move the pointer over the image in the **Detail** window. The cursor changes shape, ✥, indicating the directions in which it can be moved.

3 Press and hold the mouse button and drag the image in the **Detail** window.

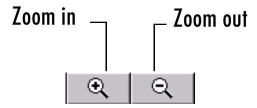
Note As you move the image in the **Detail** window, the **Detail Rectangle** in the **Overview** window moves.

Zooming In and Out on an Image

To enlarge an image to get a closer look or shrink an image to see the whole image in context, use the **Zoom** buttons on the button bar. (You can also zoom in or out on an image by changing the magnification. See "Specifying the Magnification of the Images" on page 5-21 for more information.)

To zoom in or zoom out on the base or input images:

1 Click the appropriate "magnifying glass" button.



2 Move the pointer over the image you want to zoom in or out on. The cursor changes to the cross-hair cursor + .

   You can zoom in or out on either the input or the base images, in either the **Detail** or **Overview** windows. To keep the relative magnifications of the base and input images synchronized, click the **Lock ratio** check box. See "Locking the Relative Magnification of the Input and Base Images" on page 5-22 for more information.

Note If you zoom in close on the image displayed in the **Overview** window, the **Detail Rectangle** may no longer be visible.

You can use the zoom tool in two ways:

- Position the cursor over a location in the image and click the mouse. With each click, `cpselect` changes the magnification of the image by a preset amount. (See “Specifying the Magnification of the Images” on page 5-21 for a list of some of these magnifications.) `cpselect` centers the new view of the image on the spot where you clicked.
- Alternately, you can position the cursor over a location in the image and, while pressing and holding the mouse button, draw a rectangle defining the area you want to zoom in or out on. `cpselect` magnifies the image so that the chosen section fills the **Detail** window. `cpselect` resizes the **Detail Rectangle** in the **Overview** window as well.

Note When you zoom-in or -out on an image, notice how the magnification value changes.

Specifying the Magnification of the Images

To enlarge an image to get a closer look or to shrink an image to see the whole image in context, use the magnification edit box. (You can also use the Zoom buttons to enlarge or shrink an image. See “Zooming In and Out on an Image” on page 5-20 for more information.)

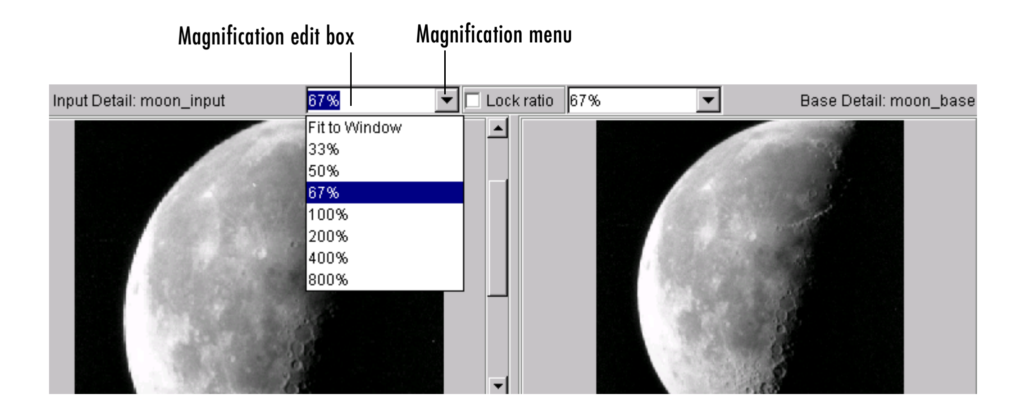
To change the magnification of an image:

1 Move the cursor into the magnification edit box of the window you want to change. The cursor changes to the text entry cursor.

Note Each **Detail** window and **Overview** window has its own magnification edit box.

---

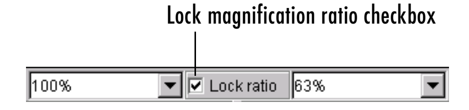
2 Type a new value in the magnification edit box and press **Enter**, or click on the menu associated with the edit box and choose from a list of preset magnifications. `cpselect` changes the magnification of the image and displays the new view in the appropriate window.



Locking the Relative Magnification of the Input and Base Images

To keep the relative magnification of the input and base images automatically synchronized in the **Detail** or **Overview** windows, click on the **Lock Ratio** checkbox. The two **Detail** windows and the two **Overview** windows each have their own **Lock ratio** checkboxes.

When the **Lock Ratio** check box is selected, `cpselect` changes the magnification of *both* the input and base images when you zoom-in or -out on either one of the images or specify a magnification value for either of the images.



Specifying Matching Control Point Pairs

The primary function of the Control Point Selection Tool is to enable you to pick control points in the image to be registered, the input image, and the image to

which you are comparing it, the base image. When you start `cpselect`, the point selection tool is enabled, by default.

You specify control points by pointing and clicking in the input and base images, in either the **Detail** or the **Overview** windows. Each point you specify in the input image must have a match in the base image. The following sections describe the ways you can use the Control Point Selection Tool to choose control point pairs:

- "Picking Control Point Pairs Manually"
- "Using Control Point Prediction" on page 5-25

This section also describes how to move control points after you've created them and how to delete control points.

Picking Control Point Pairs Manually

To specify a pair of control points in your images:

1 Click on the Control point selection button . Point selection mode is active by default.

2 Position the cursor over a feature you have visually selected in any of the images displayed. The cursor changes to a pointing finger .

   You can pick control points in either of the **Detail** windows, input or base, or in either of the **Overview** windows, input or base. You also can work in either direction: input-to-base image, or base-to-input image.

3 Click the mouse button. `cpselect` places a control point symbol at the position you specified, in both the **Detail** window and the **Overview** window. (The appearance of the control point symbol indicates its current state. Initially, control points are in an active, unmatched state. See "Control Point States" for more information.

Note Depending on where in the image you pick control points, the symbol for the point may be visible in the **Overview** window, but not in the **Detail** window.

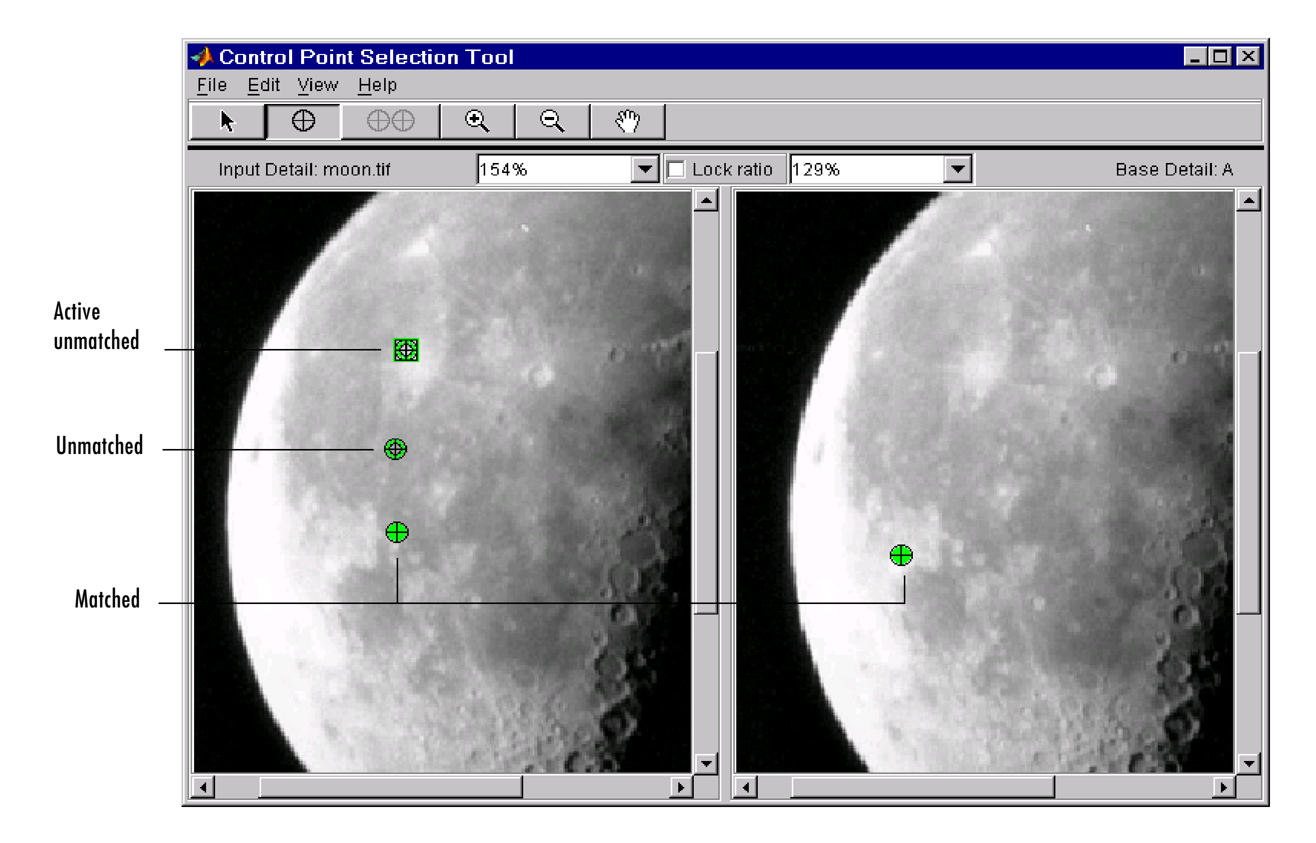
---

4 To create the match for this control point, move the cursor into the corresponding **Detail** or **Overview** window. (For example, if you started in an input window, move the cursor to a base window.)

5 Click the mouse button. `cpselect` places a control point symbol at the position you specified, in both the **Detail** and **Overview** windows. Because this control point completes a pair, the appearance of this symbol indicates an active, matched state. Note that the appearance of the first control point you selected (in step 3) also changes to an active, matched state.

You pick pairs of control points by moving from a view of the input image to a view of the base image, or vice versa. You can pick several control points in one view of the image, and then move to the corresponding window to locate their matches. To match an unmatched control point, select it to make it active, and then pick a point in the corresponding view window. When you select a match for a control point, the symbols for both points change to indicate their matched state. You can move or delete control points after you create them.

The following figure illustrates control points in several states.

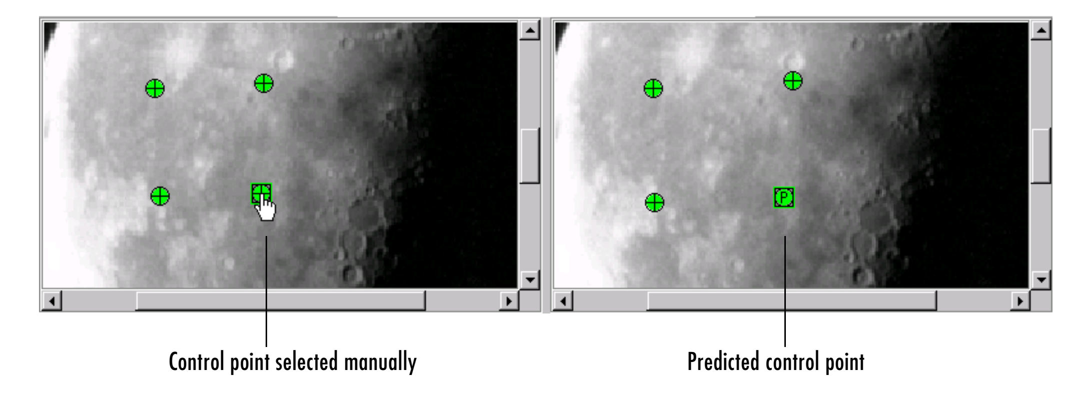


Using Control Point Prediction

Instead of picking matching control points by moving the cursor between corresponding **Detail** or **Overview** windows, you can let the Control Point Selection Tool estimate the match for the control points you specify, automatically. The Control Point Selection Tool determines the position of the matching control point based on the geometric relationship of the previously selected control points.

Note By default, the Control Point Selection Tool does not include predicted points in the set of valid control points returned in `input_points` or `base_points`. To include predicted points, you must accept them by selecting the points and fine-tuning their position with the cursor. When you move a predicted point, the Control Point Selection Tool changes the symbol to indicate that it has changed to a standard control point. For more information, see "Moving Control Points" on page 5-28.

To illustrate point prediction, this figure shows four control points selected in the input image, where the points form the four corners of a square. (The control points selections in the figure do not attempt to identify any landmarks in the image.) The figure shows the picking of a fourth point, in the left window, and the corresponding predicted point in the right window. Note how the Control Point Selection Tool places the predicted point at the same location relative to the other control points, forming the bottom right corner of the square.



Note Because the Control Point Selection Tool predicts control point locations based on the locations of the previous control points, you cannot use point prediction until you have a minimum of two pairs of matched points. Until this minimum is met, the point prediction button is disabled.

To use control point prediction:

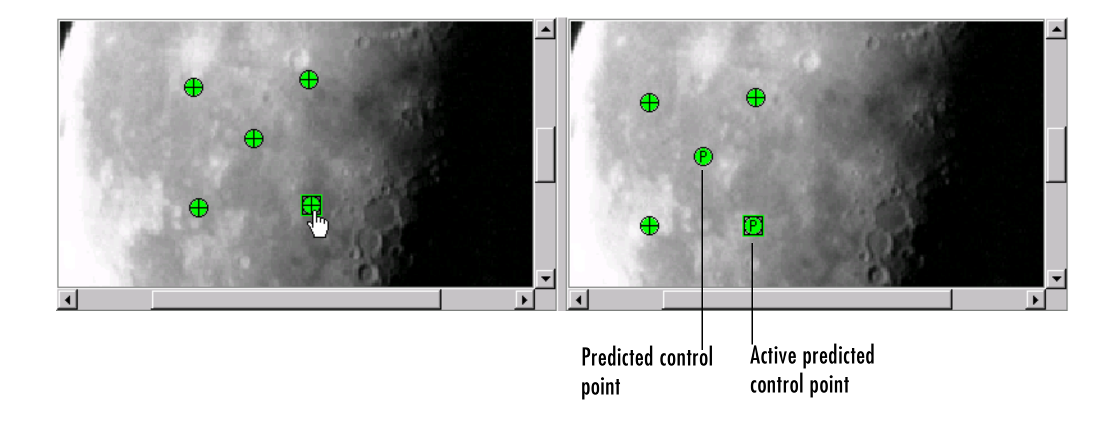
1 Click on the Control point prediction button ⊕⊕ .

2 Position the cursor anywhere in any of the images displayed. The cursor changes to a pointing finger .

You can pick control points in either of the **Detail** windows, input or base, or in either of the **Overview** windows, input or base. You also can work in either direction: input-to-base image or base-to-input image.

3 Click either mouse button. The Control Point Selection Tool places a control point symbol at the position you specified and places another control point symbol for a matching point in all the other windows. The symbol for the predicted point contains the letter "P", indicating that it's a predicted control point.

This figure illustrates predicted points in active unmatched, matched, and predicted states. For a complete description of all point states, see "Control Point States" on page 5-27.



Control Point States

The appearance of control point symbols indicate their current state. When you first pick a control point, its state is active and unmatched. When you pick the

match for a control point, the appearance of both symbols changes to indicate their matched status.

This table lists all the possible control point states with their symbols. `cpselect` displays this list in a separate window called a **Legend**. The Legend is visible by default, but you can control it's visibility using the **Legend** option from the **View** menu.

Table 5-1: Control Point States

Symbol	State	Description
	Active unmatched	The point is currently selected but does not have a matching point. This is the initial state of most points.
	Active matched	The point is currently selected and has a matching point.
	Active predicted	The point is a predicted point. If you move its position, the point changes to active matched state.
	Unmatched	The point is not selected and it is unmatched. You must select it before you can create its matching point.
	Matched	The point has a matching point.
	Predicted	This point was added by `cpselect` during point prediction.

Moving Control Points

To move a control point

1. Click on the Control point selection button or the default cursor button .
2. Position the cursor over the control point you want to move.
3. Press and hold the mouse button and drag the control point. The state of the control point changes to active when you click on it.

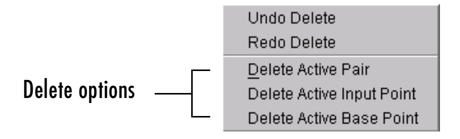
If you move a predicted control point, the state of the control point changes to a regular (nonpredicted) control point.

Deleting Control Points

To delete a control point, and optionally its matching point:

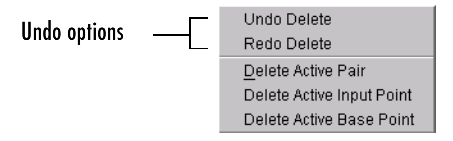
1. Click on the Control point selection button or the default cursor button .
2. Click on the control point you want to delete. Its state changes to active. If the control point has a match, both points become active.
3. Delete the point (or points) using one of these methods:
   - Pressing the **Backspace** key
   - Pressing the **Delete** key
   - Choosing one of the delete options from the **Edit** menu.

     Using this menu you can delete individual points or pairs of matched points, in the input or base images.



Undoing and Redoing Control Point Selections

You can undo a deletion or series of deletions using the **Undo Delete** option on the `cpselect` **Edit** menu.



After undoing a deletion, you can delete the points again using the **Redo** option, also on the **Edit** menu.

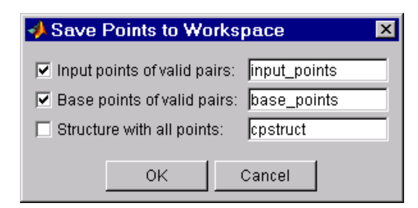
Saving Control Points

After you specify control point pairs, you must save them in the MATLAB workspace to make them available for the next step in image registration, processing by `cp2tform`.

To save control points to the MATLAB workspace:

1 Select **File** on the Control Point Selection Tool menu bar.

2 Choose the **Save Points to Workspace** option. The Control Point Selection Tool displays this dialog box:



By default, the Control Point Selection Tool saves the *x* and *y* coordinates that specify the locations of the control points you selected in two arrays named `input_points` and `base_points`, although you can specify other names. These are *n*-by-2 arrays, where *n* is the number of valid control point pairs you selected. For example, this is an example of the input_points array if you picked four pairs of control points. The values in the left column represent the *x* coordinates; the values in the right column represent the *y* coordinates.

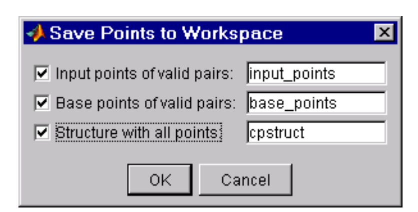
```
input_points =

  215.6667  262.3333
  225.7778  311.3333
  156.5556  340.1111
  270.8889  368.8889
```

Whenever you exit the Control Point Selection Tool, it asks if you want to save your control points.

Saving Your Control Point Selection Session

To save the current state of the Control Point Selection Tool, select the **Structure with all points** check box in the **Save Points to Workspace** dialog box.



This option saves the position of all the control points you specified and their current state in a `cpstruct` structure.

```
cpstruct =

          inputPoints: [4x2 double]
           basePoints: [4x2 double]
       inputBasePairs: [4x2 double]
                  ids: [4x1 double]
         inputIdPairs: [4x2 double]
          baseIdPairs: [4x2 double]
    isInputPredicted: [4x1 double]
     isBasePredicted: [4x1 double]
```

You can use the `cpstruct` to restart a control point selection session at the point where you left off.

This option is useful if you are picking many points over a long time and want to preserve unmatched and predicted points when you resume work. The Control Point Selection Tool does not include unmatched and predicted points in the `input_points` and `base_points` arrays.

To extract the arrays of valid control point coordinates from a `cpstruct`, use the `cpstruct2pairs` function.

Using Correlation to Improve Control Points

You may want to fine-tune the control points you selected using cpselect. Points selected by eye using the interactive tool can sometimes be improved using cross-correlation.

To use cross-correlation, pass sets of control points in the input and base images, along with the images themselves, to the cpcorr function.

```
input_pts_adj= cpcorr(input_points, base_points, input, base);
```

The cpcorr function defines 11-by-11 regions around each control point in the input image and around the matching control point in the base image, and then calculates the correlation between the values at each pixel in the region. Next, the cpcorr function looks for the position with the highest correlation value and uses this as the optimal position of the control point. The cpcorr function only moves control points up to 4 pixels based on the results of the cross correlation.

Note Features in the two images must be at the same scale and have the same orientation. They cannot be rotated relative to each other.

If cpcorr cannot correlate some of the control points, it returns their values in input_points, unmodified.

6

Neighborhood and Block Operations

This section discusses these generic block processing functions. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Block operation	An operation in which an image is processed in blocks rather than all at once. The blocks have the same size across the image. Some operation is applied to one block at a time. The blocks are reassembled to form an output image.
Border padding	Additional rows and columns temporarily added to the border(s) of an image when some of the blocks extend outside the image. The additional rows and columns normally contain zeros.
Center pixel	The pixel at the center of a neighborhood.
Column processing	An operation in which neighborhoods are reshaped into columns before processing in order to speed up computation time.
Distinct block operation	A block operation in which the blocks do not overlap.
Neighborhood operation	An operation in which each output pixel is computed from a set of neighboring input pixels. Convolution, dilation, and median filtering are examples of neighborhood operations. A neighborhood operation can also be called a sliding neighborhood operation.
Overlap	Extra rows and columns of pixels outside a block whose values are taken into account when processing the block. These extra pixels cause distinct blocks to overlap one another. The `blkproc` function enables you to specify an overlap.

Block Processing Operations

Certain image processing operations involve processing an image in sections called *blocks,* rather than processing the entire image at once.

The Image Processing Toolbox provides several functions for specific operations that work with blocks, for example, the `imdilate` function for image dilation. In addition, the toolbox provides more generic functions for processing an image in blocks. This chapter discusses these generic block processing functions.

To use one of the functions, you supply information about the size of the blocks, and specify a separate function to use to process the blocks. The block processing function does the work of breaking the input image into blocks, calling the specified function for each block, and reassembling the results into an output image.

Types of Block Processing Operations

Using these functions, you can perform various block processing operations, including *sliding neighborhood operations* and *distinct block operations*.

- In a sliding neighborhood operation, the input image is processed in a pixelwise fashion. That is, for each pixel in the input image, some operation is performed to determine the value of the corresponding pixel in the output image. The operation is based on the values of a block of neighboring pixels.
- In a distinct block operation, the input image is processed a block at a time. That is, the image is divided into rectangular blocks, and some operation is performed on each block individually to determine the values of the pixels in the corresponding block of the output image.

In addition, the toolbox provides functions for *column processing operations.* These operations are not actually distinct from block operations; instead, they are a way of speeding up block operations by rearranging blocks into matrix columns.

Note that even if you do not use these block processing functions, the information here may be useful to you, as it includes concepts fundamental to many areas of image processing. In particular, the discussion of sliding neighborhood operations is applicable to linear filtering and morphological

operations. See Chapter 7, “Linear Filtering and Filter Design” and Chapter 9, “Morphological Operations” for information about these applications.

Sliding Neighborhood Operations

A sliding neighborhood operation is an operation that is performed a pixel at a time, with the value of any given pixel in the output image being determined by applying some algorithm to the values of the corresponding input pixel's *neighborhood*. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel, which is called the *center pixel*. The neighborhood is a rectangular block, and as you move from one element to the next in an image matrix, the neighborhood block slides in the same direction.

The following figure shows the neighborhood blocks for some of the elements in a 6-by-5 matrix with 2-by-3 sliding blocks. The center pixel for each neighborhood is marked with a dot.

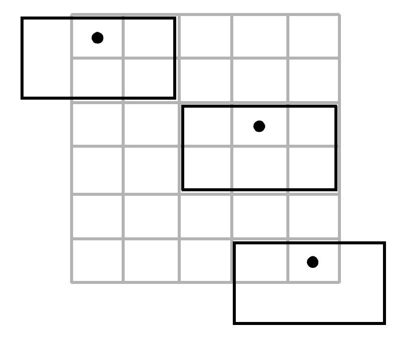


Figure 6-1: Neighborhood Blocks in a 6-by-5 Matrix

The center pixel is the actual pixel in the input image being processed by the operation. If the neighborhood has an odd number of rows and columns, the center pixel is actually in the center of the neighborhood. If one of the dimensions has even length, the center pixel is just to the left of center or just above center. For example, in a 2-by-2 neighborhood, the center pixel is the upper left one.

For any `m`-by-`n` neighborhood, the center pixel is

```
floor(([m n]+1)/2)
```

In the 2-by-3 block shown in Figure 6-1, the center pixel is (1,2), or, the pixel in the second column of the top row of the neighborhood.

To perform a sliding neighborhood operation:

1 Select a single pixel.

2 Determine the pixel's neighborhood.

3 Apply a function to the values of the pixels in the neighborhood. This function must return a scalar.

4 Find the pixel in the output image whose position corresponds to that of the center pixel in the input image. Set this output pixel to the value returned by the function.

5 Repeat steps 1 through 4 for each pixel in the input image.

For example, suppose Figure 6-1 represents an averaging operation. The function might sum the values of the six neighborhood pixels and then divide by 6. The result is the value of the output pixel.

Padding Borders

As Figure 6-1 shows, some of the pixels in a neighborhood may be missing, especially if the center pixel is on the border of the image. Notice that in the figure, the upper left and bottom right neighborhoods include "pixels" that are not part of the image.

To process these neighborhoods, sliding neighborhood operations *pad* the borders of the image, usually with 0's. In other words, these functions process the border pixels by assuming that the image is surrounded by additional rows and columns of 0's. These rows and columns do not become part of the output image and are used only as parts of the neighborhoods of the actual pixels in the image.

Linear and Nonlinear Filtering

You can use sliding neighborhood operations to implement many kinds of filtering operations. One example of a sliding neighbor operation is convolution, which is used to implement linear filtering. MATLAB provides the `conv` and `filter2` functions for performing convolution, and the toolbox provides the `imfilter` function. See Chapter 7, "Linear Filtering and Filter Design" for more information about these functions.

In addition to convolution, there are many other filtering operations you can implement through sliding neighborhoods. Many of these operations are nonlinear in nature. For example, you can implement a sliding neighborhood operation where the value of an output pixel is equal to the standard deviation of the values of the pixels in the input pixel's neighborhood.

You can use the `nlfilter` function to implement a variety of sliding neighborhood operations. `nlfilter` takes as input arguments an image, a neighborhood size, and a function that returns a scalar, and returns an image of the same size as the input image. The value of each pixel in the output image is computed by passing the corresponding input pixel's neighborhood to the function. For example, this call computes each output pixel by taking the standard deviation of the values of the input pixel's 3-by-3 neighborhood (that is, the pixel itself and its eight contiguous neighbors).

```
I2 = nlfilter(I,[3 3],'std2');
```

You can write an M-file to implement a specific function, and then use this function with `nlfilter`. For example, this command processes the matrix `I` in 2-by-3 neighborhoods with a function called `myfun.m`.

```
nlfilter(I,[2 3],@myfun);
```

`@myfun` is an example of a function handle. You can also use an inline function. For example,

```
f = inline('sqrt(min(x(:)))');
I2 = nlfilter(I,[2 2],f);
```

The example below uses `nlfilter` to set each pixel to the maximum value in its 3-by-3 neighborhood.

```
I = imread('tire.tif');
f = inline('max(x(:))');
I2 = nlfilter(I,[3 3],f);
imshow(I);
figure, imshow(I2);
```





Figure 6-2: Each Output Pixel Set to Maximum Input Neighborhood Value

Many operations that `nlfilter` can implement run much faster if the computations are performed on matrix columns rather than rectangular neighborhoods. For information about this approach, see the reference page for `colfilt`.

Note `nlfilter` is an example of a "function function." For more information on how to use this kind of function, see Function Functions in the MATLAB documentation. For more information on inline functions, see `inline` in the MATLAB Function Reference. For more information on function handles, see function_handle in the MATLAB Function Reference.

Distinct Block Operations

Distinct blocks are rectangular partitions that divide a matrix into m-by-n sections. Distinct blocks overlay the image matrix starting in the upper-left corner, with no overlap. If the blocks don't fit exactly over the image, the toolbox adds zero padding so that they do. Figure 6-3 shows a 15-by-30 matrix divided into 4-by-8 blocks.

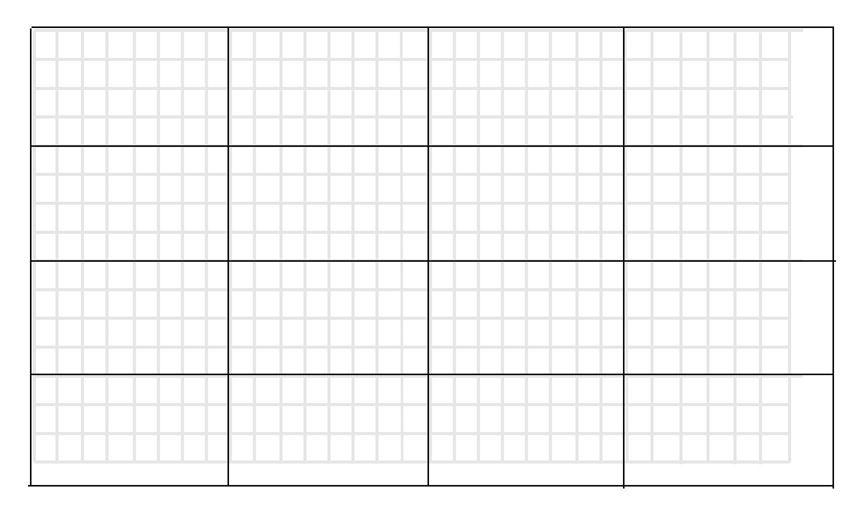


Figure 6-3: An Image Divided into Distinct Blocks

The zero padding process adds 0's to the bottom and right of the image matrix, as needed. After zero padding, the matrix is size 16-by-32.

The function blkproc performs distinct block operations. blkproc extracts each distinct block from an image and passes it to a function you specify. blkproc assembles the returned blocks to create an output image.

For example, the command below processes the matrix I in 4-by-6 blocks with the function myfun.

```
I2 = blkproc(I,[4 6],@myfun);
```

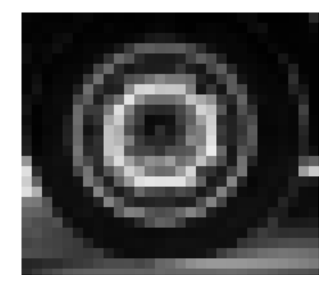
You can specify the function as an inline function. For example,

```
f =  inline('mean2(x)*ones(size(x))');
I2 = blkproc(I,[4 6],f);
```

The example below uses blkproc to set every pixel in each 8-by-8 block of an image matrix to the average of the elements in that block.

```
I = imread('tire.tif');
f = inline('uint8(round(mean2(x)*ones(size(x))))');
I2 = blkproc(I,[8 8],f);
imshow(I)
figure, imshow(I2);
```





Notice that inline computes the mean of the block and then multiplies the result by a matrix of ones, so that the output block is the same size as the input block. As a result, the output image is the same size as the input image. blkproc does not require that the images be the same size; however, if this is the result you want, you must make sure that the function you specify returns blocks of the appropriate size.

Note blkproc is an example of a "function function." For more information on how to use this kind of function, see the Function Functions section in the MATLAB documentation.

Overlap

When you call blkproc to define distinct blocks, you can specify that the blocks overlap each other, that is, you can specify extra rows and columns of pixels outside the block whose values are taken into account when processing the block. When there is an overlap, blkproc passes the expanded block (including the overlap) to the specified function.

Figure 6-4 shows the overlap areas for some of the blocks in a 15-by-30 matrix with 1-by-2 overlaps. Each 4-by-8 block has a one-row overlap above and below, and a two-column overlap on each side. In the figure, shading indicates the overlap. The 4-by-8 blocks overlay the image matrix starting in the upper-left corner.

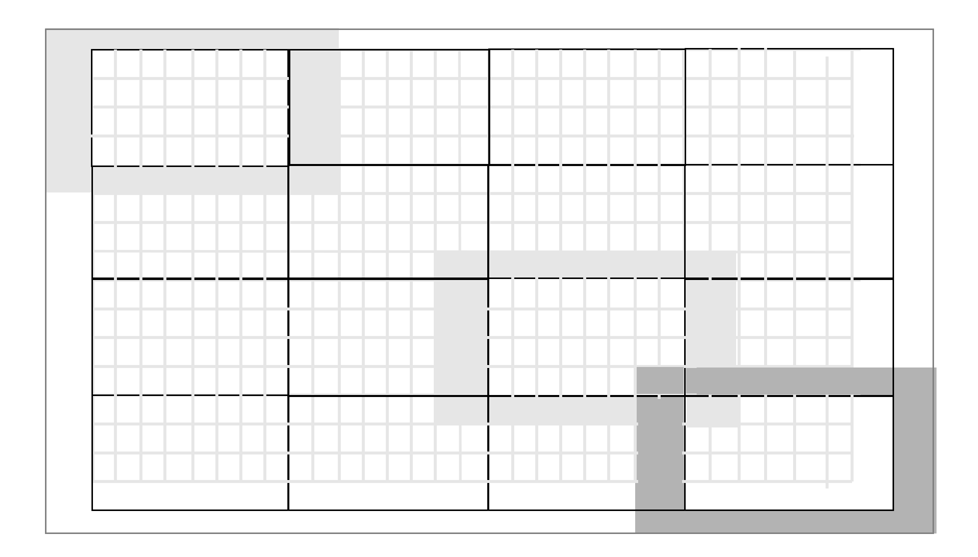


Figure 6-4: An Image Divided into Distinct Blocks With Specified Overlaps

To specify the overlap, you provide an additional input argument to `blkproc`. To process the blocks in the figure above with the function `myfun`, the call is

```
B = blkproc(A,[4 8],[1 2],@myfun)
```

Overlap often increases the amount of zero padding needed. For example, in Figure 6-3, the original 15-by-30 matrix became a 16-by-32 matrix with zero padding. When the 15-by-30 matrix includes a 1-by-2 overlap, the padded matrix becomes an 18-by-36 matrix. The outermost rectangle in the figure delineates the new boundaries of the image after padding has been added to accommodate the overlap plus block processing. Notice that in the figure above, padding has been added to the left and top of the original image, not just to the right and bottom.

Column Processing

The toolbox provides functions that you can use to process sliding neighborhoods or distinct blocks as columns. This approach is useful for operations that MATLAB performs columnwise; in many cases, column processing can reduce the execution time required to process an image.

For example, suppose the operation you are performing involves computing the mean of each block. This computation is much faster if you first rearrange the blocks into columns, because you can compute the mean of every column with a single call to the `mean` function, rather than calling `mean` for each block individually.

You can use the `colfilt` function to implement column processing. This function:

1. Reshapes each sliding or distinct block of an image matrix into a column in a temporary matrix
2. Passes the temporary matrix to a function you specify
3. Rearranges the resulting matrix back into the original shape

Sliding Neighborhoods

For a sliding neighborhood operation, `colfilt` creates a temporary matrix that has a separate column for each pixel in the original image. The column corresponding to a given pixel contains the values of that pixel's neighborhood from the original image.

Figure 6-5 illustrates this process. In this figure, a 6-by-5 image matrix is processed in 2-by-3 neighborhoods. `colfilt` creates one column for each pixel in the image, so there are a total of 30 columns in the temporary matrix. Each pixel's column contains the value of the pixels in its neighborhood, so there are six rows. `colfilt` zero pads the input image as necessary. For example, the neighborhood of the upper left pixel in the figure has two zero-valued neighbors, due to zero padding.

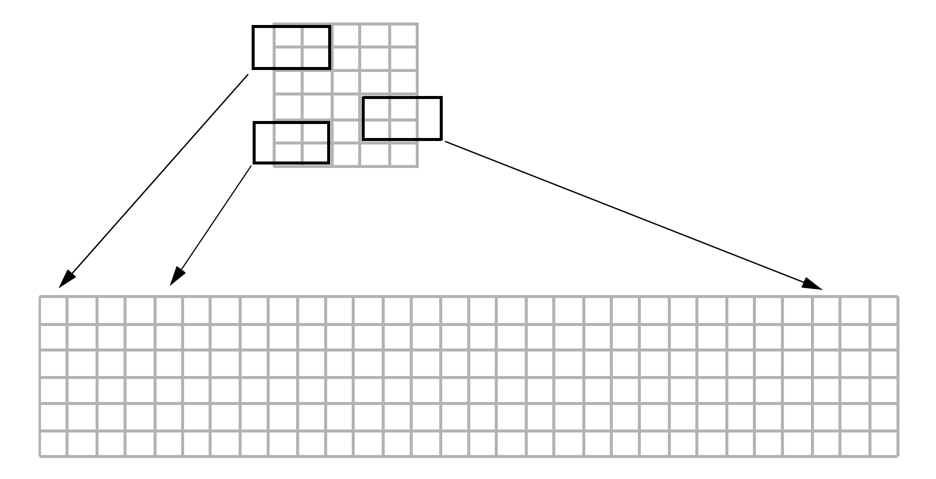


Figure 6-5: colfilt Creates a Temporary Matrix for Sliding Neighborhood

The temporary matrix is passed to a function, which must return a single value for each column. (Many MATLAB functions work this way, for example, `mean`, `median`, `std`, `sum`, etc.) The resulting values are then assigned to the appropriate pixels in the output image.

`colfilt` can produce the same results as `nlfilter` with faster execution time; however, it may use more memory. The example below sets each output pixel to the maximum value in the input pixel's neighborhood, producing the same result as the `nlfilter` example shown in Figure 6-2.

```
I2 = colfilt(I,[3 3],'sliding',@max);
```

Distinct Blocks

For a distinct block operation, `colfilt` creates a temporary matrix by rearranging each block in the image into a column. `colfilt` pads the original image with 0's, if necessary, before creating the temporary matrix.

Figure 6-6 illustrates this process. In this figure, a 6-by-16 image matrix is processed in 4-by-6 blocks. `colfilt` first zero pads the image to make the size 8-by-18 (six 4-by-6 blocks), and then rearranges the blocks into 6 columns of 24 elements each.

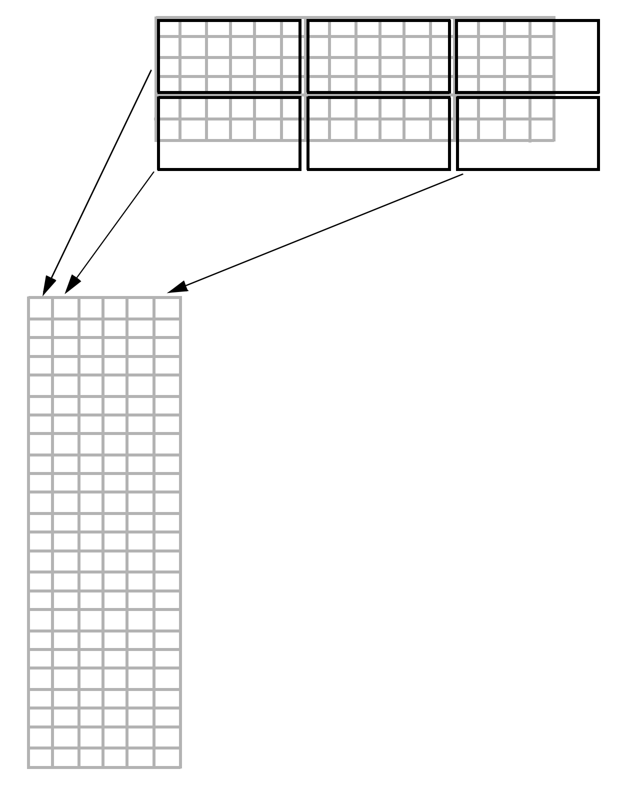


Figure 6-6: colfilt Creates a Temporary Matrix for Distinct Block Operation

After rearranging the image into a temporary matrix, `colfilt` passes this matrix to the function. The function must return a matrix of the same size as the temporary matrix. If the block size is `m`-by-`n`, and the image is `mm`-by-`nn`, the size of the temporary matrix is `(m*n)`-by-`(ceil(mm/m)*ceil(nn/n))`. After the function processes the temporary matrix, the output is rearranged back into the shape of the original image matrix.

This example sets all the pixels in each 8-by-8 block of an image to the mean pixel value for the block, producing the same result as the `blkproc` example in "Distinct Block Operations" on page 6-9.

```
I = im2double(imread('tire.tif'));
f = inline('ones(64,1)*mean(x)');
I2 = colfilt(I,[8 8],'distinct',f);
```

Notice that the inline function computes the mean of the block and then multiplies the result by a vector of ones, so that the output block is the same size as the input block. As a result, the output image is the same size as the input image.

Restrictions

You can use `colfilt` to implement many of the same distinct block operations that `blkproc` performs. However, `colfilt` has certain restrictions that `blkproc` does not:

- The output image must be the same size as the input image.
- The blocks cannot overlap.

For situations that do not satisfy these constraints, use `blkproc`.

7

Linear Filtering and Filter Design

The Image Processing Toolbox provides a number of functions for designing and implementing two-dimensional linear filters for image data. This section describes these functions and how to use them effectively. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter. Note that this table includes brief definitions of terms related to filter design; a detailed discussion of these terms and the theory behind filter design is outside the scope of this User's Guide.

Terms	Definitions
Convolution	A neighborhood operation in which each output pixel is a weighted sum of neighboring input pixels. The weights are defined by the convolution kernel. Image processing operations implemented with convolution include smoothing, sharpening, and edge enhancement.
Convolution kernel	A matrix of weights used to perform convolution. A convolution kernel is a correlation kernel that has been rotated 180 degrees.
Correlation	A neighborhood operation in which each output pixel is a weighted sum of neighboring input pixels. The weights are defined by the correlation kernel. Correlation is closely related mathematically to convolution.
Correlation kernel	A matrix of weights used to perform correlation. The filter design functions in the Image Processing Toolbox return correlation kernels. A correlation kernel is a convolution kernel that has been rotated 180 degrees.
FIR filter	A filter whose response to a single point, or impulse, has finite extent. FIR stands for finite impulse response. An FIR filter can be implemented using convolution. All filter design functions in the Image Processing Toolbox return FIR filters.
Frequency response	A mathematical function describing the gain of a filter in response to different input frequencies.
Neighborhood operation	An operation in which each output pixel is computed from a set of neighboring input pixels. Convolution, dilation, and median filtering are examples of neighborhood operations.

Terms	Definitions
Ripples	Oscillations around a constant value. The frequency response of a practical filter often has ripples where the frequency response of an ideal filter is flat.
Window method	A filter design method that multiples the ideal impulse response by a window function, which tapers the ideal impulse response. The resulting filter's frequency response approximates a desired frequency response.

Linear Filtering

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features.

Filtering is a *neighborhood operation,* in which the value of any given pixel in the output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel. (See Chapter 6, "Neighborhood and Block Operations", for a general discussion of neighborhood operations.)

Linear filtering is filtering in which the value of an output pixel is a linear combination of the values of the pixels in the input pixel's neighborhood.

This section discusses linear filtering in MATLAB and the Image Processing Toolbox. It includes:

- A description of filtering, using convolution and correlation
- A description of how to use the `imfilter` function to perform filtering
- A discussion about using predefined filter types

See "Filter Design" on page 7-16 for information about how to design filters.

Convolution

Linear filtering of an image is accomplished through an operation called *convolution*. In convolution, the value of an output pixel is computed as a weighted sum of neighboring pixels. The matrix of weights is called the *convolution kernel*, also known as the *filter*.

For example, suppose the image is

```
A = [17  24   1   8  15
     23   5   7  14  16
      4   6  13  20  22
     10  12  19  21   3
     11  18  25   2   9]
```

and the convolution kernel is

```
h = [8    1    6
     3    5    7
     4    9    2]
```

Then Figure 7-1 shows how to compute the (2,4) output pixel using these steps:

1. Rotate the convolution kernel 180 degrees about its center element.
2. Slide the center element of the convolution kernel so that lies on top of the (2,4) element of A.
3. Multiply each weight in the rotated convolution kernel by the pixel of A underneath.
4. Sum up the individual products from step 3.

Hence the (2,4) output pixel is

$$1 \cdot 2 + 8 \cdot 9 + 15 \cdot 4 + 7 \cdot 7 + 14 \cdot 5 + 16 \cdot 3 + 13 \cdot 6 + 20 \cdot 1 + 22 \cdot 8 = 575$$

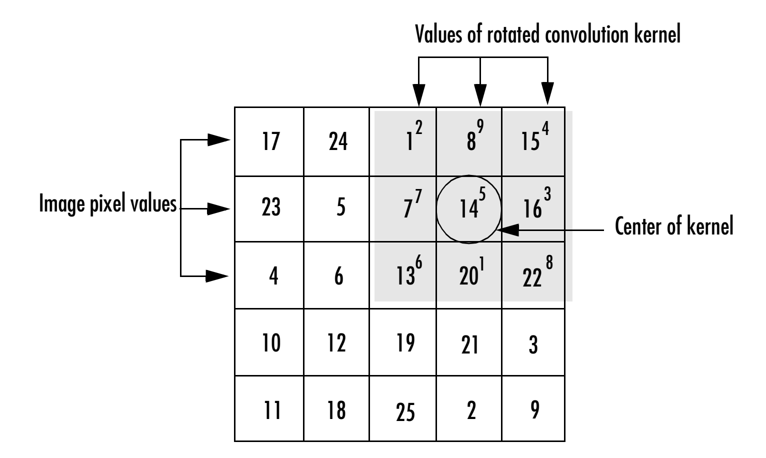


Figure 7-1: Computing the (2,4) Output of Convolution

Correlation

The operation called *correlation* is closely related to convolution. In correlation, the value of an output pixel is also computed as a weighted sum of neighboring pixels. The difference is that the matrix of weights, in this case called the *correlation kernel*, is not rotated during the computation. Figure 7-2 shows how to compute the (2,4) output pixel of the correlation of `A`, assuming `h` is correlation kernel instead of a convolution kernel, using these steps:

1. Slide the center element of the correlation kernel so that lies on top of the (2,4) element of A.

2. Multiply each weight in the correlation kernel by the pixel of `A` underneath.

3. Sum up the individual products from step 3.

The (2,4) output pixel from the correlation is

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585$$

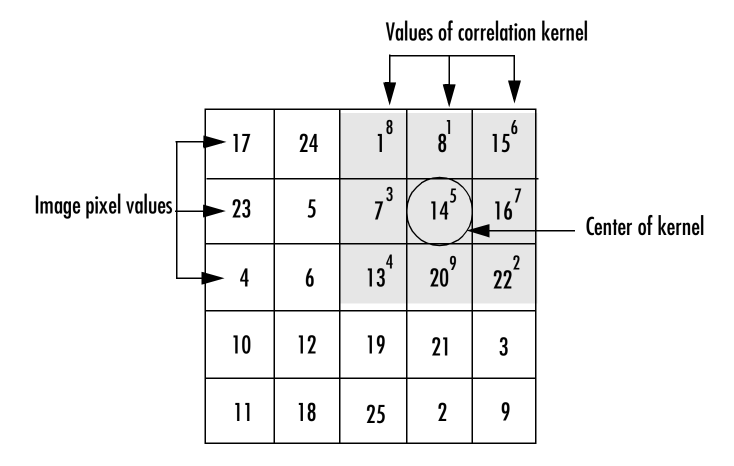
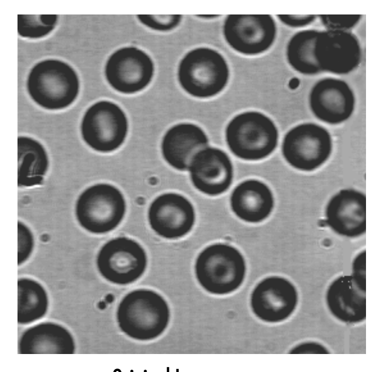


Figure 7-2: Computing the (2,4) Output of Correlation

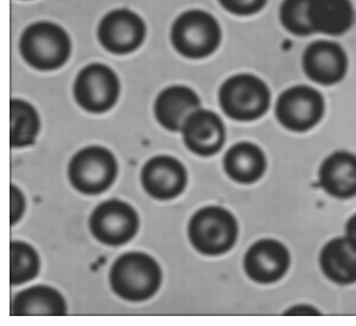
Filtering Using imfilter

Filtering of images, either by correlation or convolution, can be performed using the toolbox function `imfilter`. This example filters the image in the file `blood1.tif` with a 5-by-5 filter containing equal weights. Such a filter is often called an *averaging filter*.

```
I = imread('blood1.tif');
h = ones(5,5) / 25;
I2 = imfilter(I,h);
imshow(I), title('Original image')
figure, imshow(I2), title('Filtered image')
```



Original Image



Filtered Image

Data Types

The `imfilter` function handles data types similar to the way the image arithmetic functions do, as described in "Image Arithmetic Truncation Rules" on page 2-22. The output image has the same data type, or numeric class, as the input image. The `imfilter` function computes the value of each output pixel using double-precision, floating-point arithmetic. If the result exceeds the range of the data type, the `imfilter` function truncates the result to that data type's allowed range. If it is an integer data type, `imfilter` rounds fractional values.

Because of the truncation behavior, you may sometimes want to consider converting your image to a different data type before calling `imfilter`. In this example, the output of `imfilter` has negative values when the input is of class `double`.

```
A = magic(5)

A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

h = [-1 0 1]

h =
    -1     0     1

imfilter(A,h)

ans =
    24   -16   -16    14    -8
     5   -16     9     9   -14
     6     9    14     9   -20
    12     9     9   -16   -21
    18    14   -16   -16    -2
```

Notice that the result has negative values. Now suppose `A` was of class `uint8`, instead of `double`.

```
A = uint8(magic(5));
imfilter(A,h)

ans =

    24     0     0    14     0
     5     0     9     9     0
     6     9    14     9     0
    12     9     9     0     0
    18    14     0     0     0
```

Since the input to `imfilter` is of class `uint8`, the output also is of class `uint8`, and so the negative values are truncated to `0`. In such cases, it may be appropriate to convert the image to another type, such as a signed integer type, `single`, or `double`, before calling `imfilter`.

Correlation and Convolution Options

The `imfilter` function can perform filtering using either correlation or convolution. It uses correlation by default, because the filter design functions, described in “Filter Design” on page 7-16, and the `fspecial` function, described in “Using Predefined Filter Types” on page 7-14, produce correlation kernels.

However, if you want to perform filtering using convolution instead, you can pass the string `'conv'` as optional input argument to `imfilter`. For example,

```
A = magic(5);
h = [-1 0 1]
imfilter(A,h)   % filter using correlation

ans =
    24   -16   -16    14    -8
     5   -16     9     9   -14
     6     9    14     9   -20
    12     9     9   -16   -21
    18    14   -16   -16    -2

imfilter(A,h,'conv')   % filter using convolution

ans =

   -24    16    16   -14     8
    -5    16    -9    -9    14
    -6    -9   -14    -9    20
   -12    -9    -9    16    21
   -18   -14    16    16     2
```

Boundary Padding Options

When computing an output pixel at the boundary of an image, a portion of the convolution or correlation kernel is usually off the edge of the image, as illustrated in the figure below.

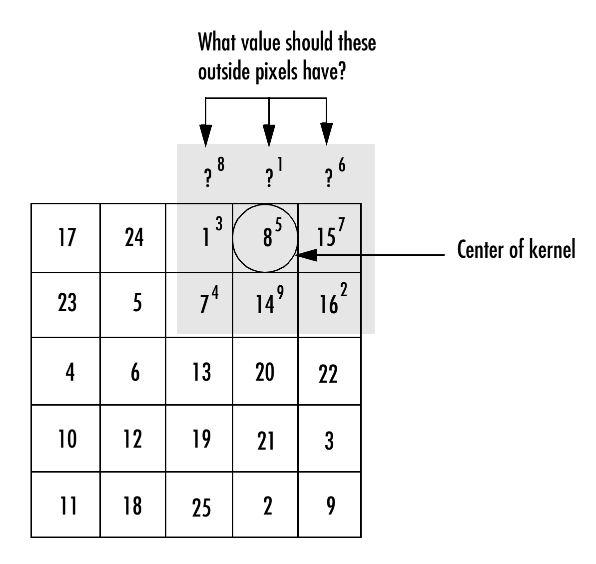


Figure 7-3: When the Values of the Kernel Fall Outside the Image

The `imfilter` function normally fills in these "off-the-edge" image pixels by assuming that they are `0`. This is called zero-padding and is illustrated in the figure below.

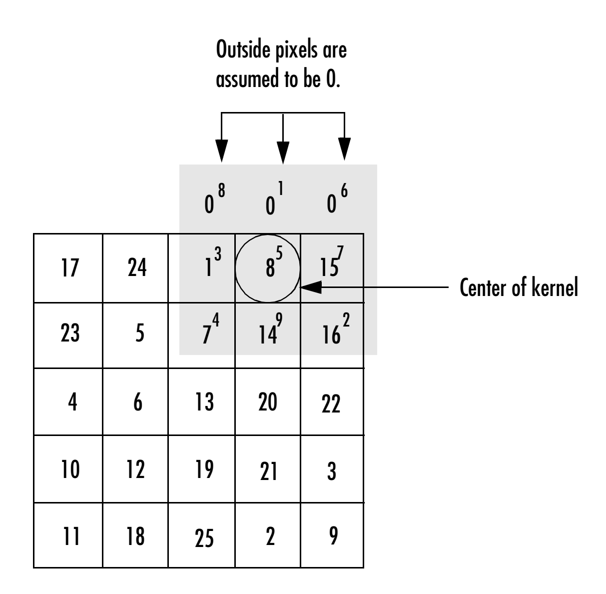
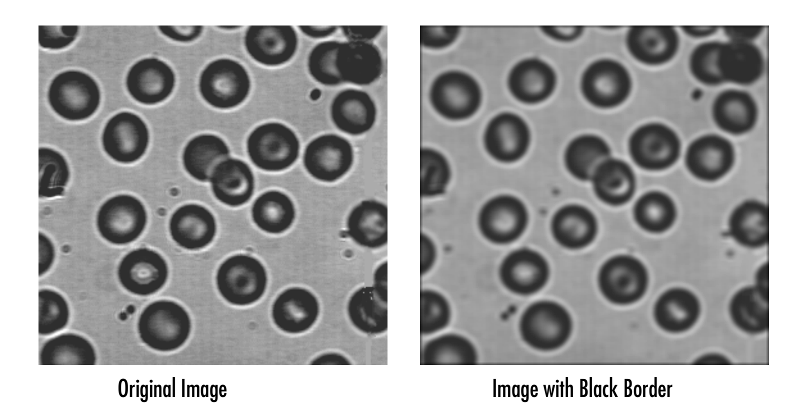


Figure 7-4: Zero-Padding of Outside Pixels

When filtering an image, zero-padding can result in a dark band around the edge of the image, as shown in this example.

```
I = imread('blood1.tif');
h = ones(5,5)/25;
I2 = imfilter(I,h);
imshow(I), title('Original image')
figure, imshow(I2), title('Filtered image')
```



To eliminate the zero-padding artifacts around the edge of the image, `imfilter` offers an alternative boundary padding method called *border replication*. In border replication, the value of any pixel outside the image is determined by replicating the value from the nearest border pixel. This is illustrated in the figure below.

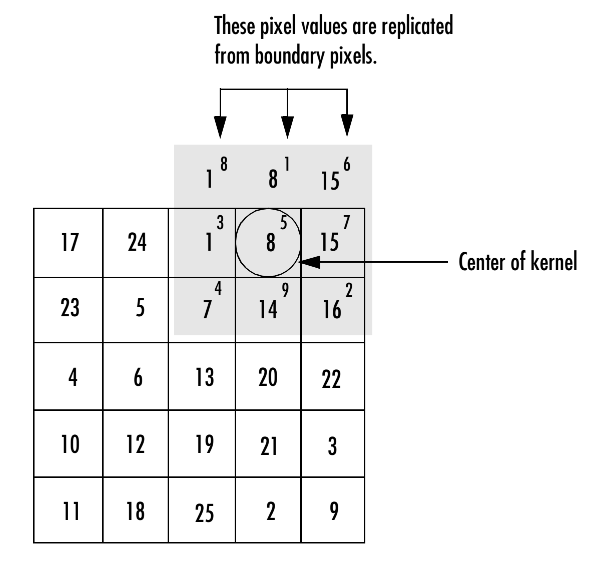


Figure 7-5: Replicated Boundary Pixels

To filter using border replication, pass the additional optional argument `'replicate'` to `imfilter`.

```
I3 = imfilter(I,h,'replicate');
figure, imshow(I3), title('Filtered with border replication')
```

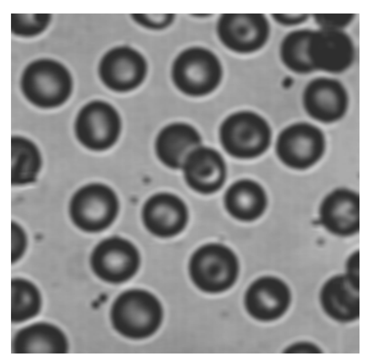


Image Border with Replication

The `imfilter` function supports other boundary padding options, such as `'circular'` and `'symmetric'`. See the reference page for `imfilter` for details.

Multidimensional Filtering

The `imfilter` function can handle both multidimensional images and multidimensional filters. A convenient property of filtering is that filtering a three-dimensional image with a two-dimensional filter is equivalent to filtering each plane of the three-dimensional image individually with the same two-dimensional filter. This property makes it easy, for example, to filter each color plane of a truecolor image with the same filter.

```
rgb = imread('flowers.tif');
h = ones(5,5) / 25;
rgb2 = imfilter(rgb,h);
imshow(rgb), title('Original image')
figure, imshow(rgb2), title('Filtered image')
```



Original Image



Filtered Image

Relationship to Other Filtering Functions

MATLAB has several two-dimensional and multidimensional filtering functions. The function `filter2` performs two-dimensional correlation; `conv2` performs two-dimensional convolution; and `convn` performs multidimensional convolution. Each of these other filtering functions always converts the input to `double`, and the output is always `double`. Also, each of these other filtering functions always assumes the input is zero-padded, and they do not support other padding options.

In contrast, the `imfilter` function does not convert input images to `double`. The `imfilter` function also offers a flexible set of boundary padding options, as described in "Boundary Padding Options" on page 7-10.

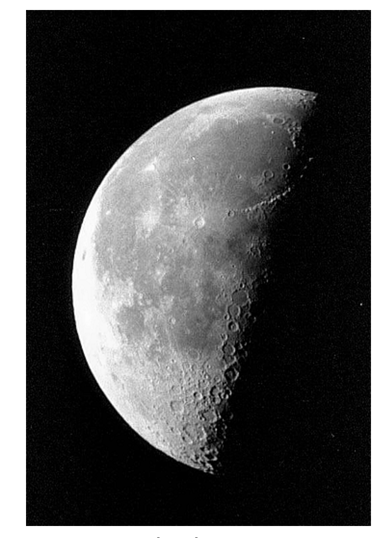
Using Predefined Filter Types

The `fspecial` function produces several kinds of predefined filters, in the form of correlation kernels. After creating a filter with `fspecial`, you can apply it directly to your image data using `imfilter`. This example illustrates applying an *unsharp masking* filter to an intensity image. The unsharp masking filter has the effect of making edges and fine detail in the image more crisp.

```
I = imread('moon.tif');
h = fspecial('unsharp');
I2 = imfilter(I,h);
imshow(I), title('Original image')
figure, imshow(I2), title('Filtered image')
```



Original Image



Filtered Image

Filter Design

This section describes working in the frequency domain to design filters. Topics discussed include:

- Finite impulse response (FIR) filters, the class of linear filter that the toolbox supports
- The frequency transformation method, which transforms a one-dimensional FIR filter into a two-dimensional FIR filter
- The frequency sampling method, which creates a filter based on a desired frequency response
- The windowing method, which multiplies the ideal impulse response with a window function to generate the filter
- Creating the desired frequency response matrix
- Computing the frequency response of a filter

This section assumes you are familiar with working in the frequency domain. This topic is discussed in many signal processing and image processing textbooks.

Note Most of the design methods described in this section work by creating a two-dimensional filter from a one-dimensional filter or window created using functions from the Signal Processing Toolbox. Although this toolbox is not required, you may find it difficult to design filters in the Image Processing Toolbox if you do not have the Signal Processing Toolbox as well.

FIR Filters

The Image Processing Toolbox supports one class of linear filter, the two-dimensional finite impulse response (FIR) filter. FIR filters have several characteristics that make them ideal for image processing in the MATLAB environment:

- FIR filters are easy to represent as matrices of coefficients.
- Two-dimensional FIR filters are natural extensions of one-dimensional FIR filters.

- There are several well-known, reliable methods for FIR filter design.
- FIR filters are easy to implement.
- FIR filters can be designed to have linear phase, which helps prevent distortion.

Another class of filter, the infinite impulse response (IIR) filter, is not as suitable for image processing applications. It lacks the inherent stability and ease of design and implementation of the FIR filter. Therefore, this toolbox does not provide IIR filter support.

Frequency Transformation Method

The frequency transformation method transforms a one-dimensional FIR filter into a two-dimensional FIR filter. The frequency transformation method preserves most of the characteristics of the one-dimensional filter, particularly the transition bandwidth and ripple characteristics. This method uses a *transformation matrix*, a set of elements that defines the frequency transformation.

The toolbox function `ftrans2` implements the frequency transformation method. This function's default transformation matrix produces filters with nearly circular symmetry. By defining your own transformation matrix, you can obtain different symmetries. (See Jae S. Lim, *Two-Dimensional Signal and Image Processing*, 1990, for details.)

The frequency transformation method generally produces very good results, as it is easier to design a one-dimensional filter with particular characteristics than a corresponding two-dimensional filter. For instance, the next example designs an optimal equiripple one-dimensional FIR filter and uses it to create a two-dimensional filter with similar characteristics. The shape of the one-dimensional frequency response is clearly evident in the two-dimensional response.

```
b = remez(10,[0 0.4 0.6 1],[1 1 0 0]);
h = ftrans2(b);
[H,w] = freqz(b,1,64,'whole');
colormap(jet(64))
plot(w/pi-1,fftshift(abs(H)))
figure, freqz2(h,[32 32])
```

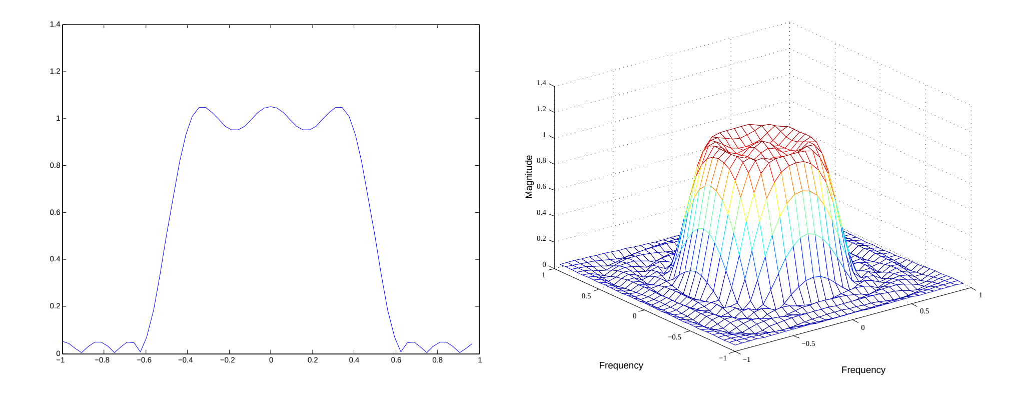


Figure 7-6: A One-Dimensional Frequency Response (left) and the Corresponding Two-Dimensional Frequency Response (right)

Frequency Sampling Method

The frequency sampling method creates a filter based on a desired frequency response. Given a matrix of points that defines the shape of the frequency response, this method creates a filter whose frequency response passes through those points. Frequency sampling places no constraints on the behavior of the frequency response between the given points; usually, the response ripples in these areas.

The toolbox function `fsamp2` implements frequency sampling design for two-dimensional FIR filters. `fsamp2` returns a filter `h` with a frequency response that passes through the points in the input matrix `Hd`. The example below creates an 11-by-11 filter using `fsamp2`, and plots the frequency response of the resulting filter. (The `freqz2` function in this example calculates the two-dimensional frequency response of a filter. See "Computing the Frequency Response of a Filter" on page 7-21 for more information.)

```
Hd = zeros(11,11); Hd(4:8,4:8) = 1;
[f1,f2] = freqspace(11,'meshgrid');
mesh(f1,f2,Hd), axis([-1 1 -1 1 0 1.2]), colormap(jet(64))
h = fsamp2(Hd);
figure, freqz2(h,[32 32]), axis([-1 1 -1 1 0 1.2])
```

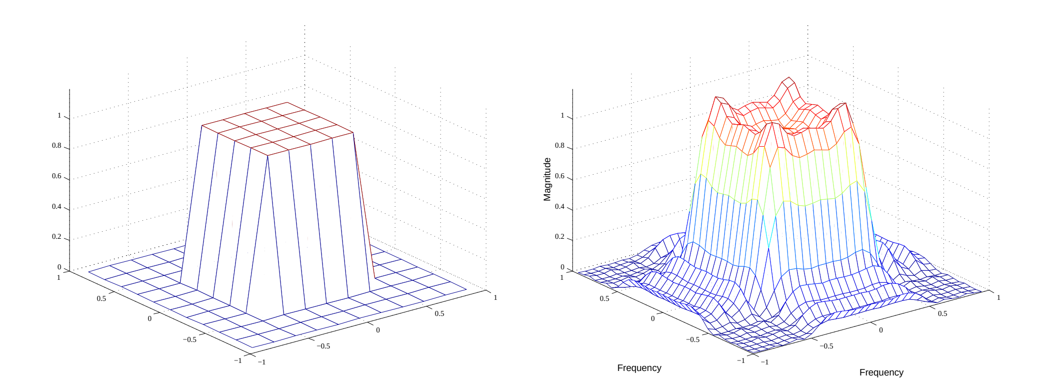


Figure 7-7: Desired Two-Dimensional Frequency Response (left) and Actual Two-Dimensional Frequency Response (right)

Notice the ripples in the actual frequency response, compared to the desired frequency response. These ripples are a fundamental problem with the frequency sampling design method. They occur wherever there are sharp transitions in the desired response.

You can reduce the spatial extent of the ripples by using a larger filter. However, a larger filter does not reduce the height of the ripples, and requires more computation time for filtering. To achieve a smoother approximation to the desired frequency response, consider using the frequency transformation method or the windowing method.

Windowing Method

The windowing method involves multiplying the ideal impulse response with a window function to generate a corresponding filter. Like the frequency sampling method, the windowing method produces a filter whose frequency response approximates a desired frequency response. The windowing method, however, tends to produce better results than the frequency sampling method.

The toolbox provides two functions for window-based filter design, `fwind1` and `fwind2`. `fwind1` designs a two-dimensional filter by using a two-dimensional window that it creates from one or two one-dimensional windows that you specify. `fwind2` designs a two-dimensional filter by using a specified two-dimensional window directly.

`fwind1` supports two different methods for making the two-dimensional windows it uses:

- Transforming a single one-dimensional window to create a two-dimensional window that is nearly circularly symmetric, by using a process similar to rotation
- Creating a rectangular, separable window from two one-dimensional windows, by computing their outer product

The example below uses `fwind1` to create an 11-by-11 filter from the desired frequency response `Hd`. Here, the `hamming` function from the *Signal Processing Toolbox* is used to create a one-dimensional window, which `fwind1` then extends to a two-dimensional window.

```
Hd = zeros(11,11); Hd(4:8,4:8) = 1;
[f1,f2] = freqspace(11,'meshgrid');
mesh(f1,f2,Hd), axis([-1 1 -1 1 0 1.2]), colormap(jet(64))
h = fwind1(Hd,hamming(11));
figure, freqz2(h,[32 32]), axis([-1 1 -1 1 0 1.2])
```

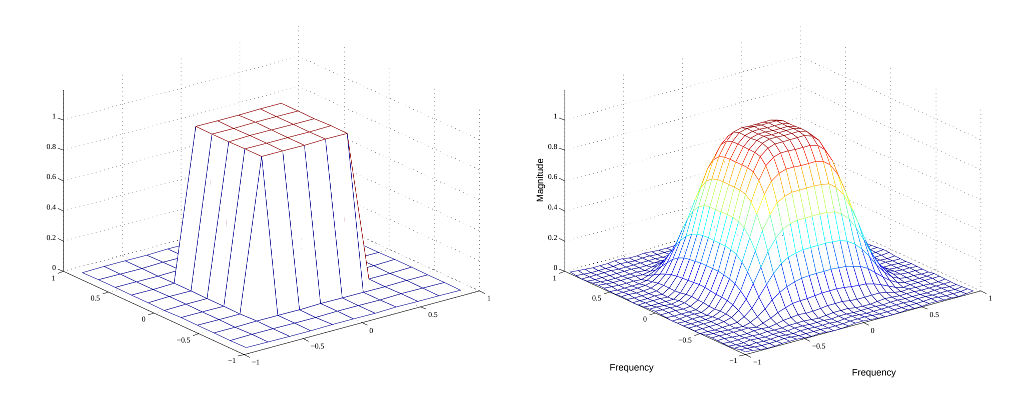


Figure 7-8: Desired Two-Dimensional Frequency Response (left) and Actual Two-Dimensional Frequency Response (right)

Creating the Desired Frequency Response Matrix

The filter design functions `fsamp2`, `fwind2`, and `fwind2` all create filters based on a desired frequency response magnitude matrix. You can create an appropriate desired frequency response matrix using the `freqspace` function.

`freqspace` returns correct, evenly spaced frequency values for any size response. If you create a desired frequency response matrix using frequency points other than those returned by `freqspace`, you may get unexpected results, such as nonlinear phase.

For example, to create a circular ideal lowpass frequency response with cutoff at 0.5 use

```
[f1,f2] = freqspace(25,'meshgrid');
Hd = zeros(25,25); d = sqrt(f1.^2 + f2.^2) < 0.5;
Hd(d) = 1;
mesh(f1,f2,Hd)
```

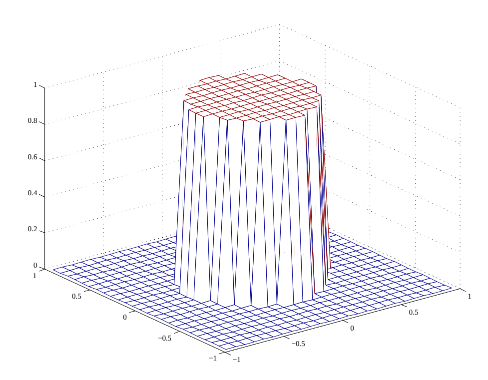


Figure 7-9: Ideal Circular Lowpass Frequency Response

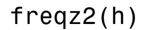
Note that for this frequency response, the filters produced by `fsamp2`, `fwind1`, and `fwind2` are real. This result is desirable for most image processing applications. To achieve this in general, the desired frequency response should be symmetric about the frequency origin (`f1 = 0`, `f2 = 0`).

Computing the Frequency Response of a Filter

The `freqz2` function computes the frequency response for a two-dimensional filter. With no output arguments, `freqz2` creates a mesh plot of the frequency response. For example, consider this FIR filter,

```
h =[0.1667    0.6667    0.1667
    0.6667   -3.3333    0.6667
    0.1667    0.6667    0.1667];
```

This command computes and displays the 64-by-64 point frequency response of h.



```
freqz2(h)
```

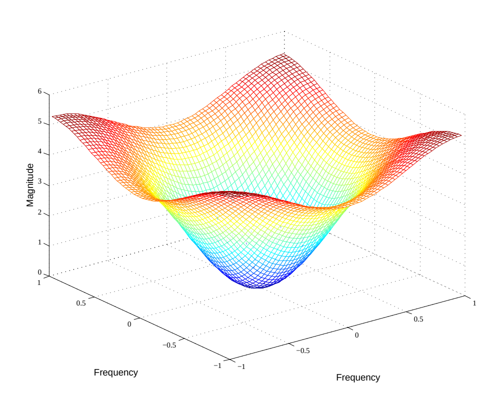


Figure 7-10: The Frequency Response of a Two-Dimensional Filter

To obtain the frequency response matrix H and the frequency point vectors f1 and f2, use output arguments

```
[H,f1,f2] = freqz2(h);
```

freqz2 normalizes the frequencies f1 and f2 so that the value 1.0 corresponds to half the sampling frequency, or π radians.

For a simple m-by-n response, as shown above, freqz2 uses the two-dimensional fast Fourier transform function fft2. You can also specify vectors of arbitrary frequency points, but in this case freqz2 uses a slower algorithm.

See "Fourier Transform" on page 8-3 for more information about the fast Fourier transform and its application to linear filtering and filter design.

Transforms

The usual mathematical representation of an image is a function of two spatial variables: $f(x, y)$. The value of the function at a particular location (x, y) represents the intensity of the image at that point. The term *transform* refers to an alternative mathematical representation of an image.

This section defines several important transforms and shows examples of their application to image processing. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter. Note that this table includes brief definitions of terms related to transforms; a detailed discussion of these terms and the theory behind transforms is outside the scope of this User's Guide.

Terms	Definitions
Discrete transform	A transform whose input and output values are discrete samples, making it convenient for computer manipulation. Discrete transforms implemented by MATLAB and the Image Processing Toolbox include the discrete Fourier transform (DFT) and the discrete cosine transform (DCT).
Frequency domain	The domain in which an image is represented by a sum of periodic signals with varying frequency.
Inverse transform	An operation that when performed on a transformed image, produces the original image.
Spatial domain	The domain in which an image is represented by intensities at given points in space. This is the most common representation for image data.
Transform	An alternative mathematical representation of an image. For example, the Fourier transform is a representation of an image as a sum of complex exponentials of varying magnitudes, frequencies, and phases. Transforms are useful for a wide range of purposes, including convolution, enhancement, feature detection, and compression.

Fourier Transform

The Fourier transform is a representation of an image as a sum of complex exponentials of varying magnitudes, frequencies, and phases. The Fourier transform plays a critical role in a broad range of image processing applications, including enhancement, analysis, restoration, and compression.

This section includes the following subsections:

- "Definition of Fourier Transform"
- "Discrete Fourier Transform" on page 8-8, including a discussion of fast Fourier transform
- "Applications" on page 8-11 (sample applications using Fourier transforms)

Definition of Fourier Transform

If $f(m, n)$ is a function of two discrete spatial variables m and n, then we define the *two-dimensional Fourier transform* of $f(m, n)$ by the relationship

$$F(\omega_1, \omega_2) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n) e^{-j\omega_1 m} e^{-j\omega_2 n}$$

The variables ω_1 and ω_2 are frequency variables; their units are radians per sample. $F(\omega_1, \omega_2)$ is often called the *frequency-domain* representation of $f(m, n)$. $F(\omega_1, \omega_2)$ is a complex-valued function that is periodic both in ω_1 and ω_2, with period 2π. Because of the periodicity, usually only the range $-\pi \leq \omega_1, \omega_2 \leq \pi$ is displayed. Note that $F(0, 0)$ is the sum of all the values of $f(m, n)$. For this reason, $F(0, 0)$ is often called the *constant component* or *DC component* of the Fourier transform. (DC stands for direct current; it is an electrical engineering term that refers to a constant-voltage power source, as opposed to a power source whose voltage varies sinusoidally.)

The inverse two-dimensional Fourier transform is given by

$$f(m, n) = \frac{1}{4\pi^2} \int_{\omega_1=-\pi}^{\pi} \int_{\omega_2=-\pi}^{\pi} F(\omega_1, \omega_2) e^{j\omega_1 m} e^{j\omega_2 n} \, d\omega_1 \, d\omega_2$$

Roughly speaking, this equation means that $f(m, n)$ can be represented as a sum of an infinite number of complex exponentials (sinusoids) with different

frequencies. The magnitude and phase of the contribution at the frequencies (ω_1, ω_2) are given by $F(\omega_1, \omega_2)$.

Example

Consider a function $f(m, n)$ that equals 1 within a rectangular region and 0 everywhere else.

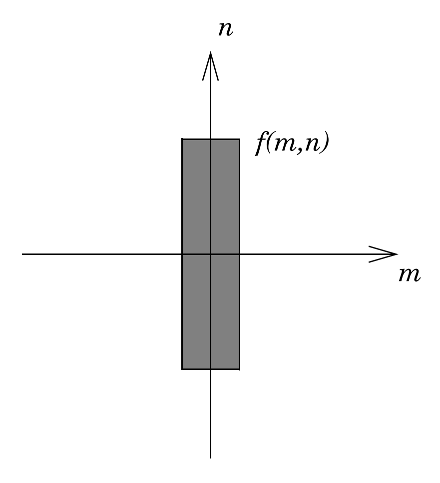


Figure 8-1: A Rectangular Function

To simplify the diagram, $f(m, n)$ is shown as a continuous function, even though the variables m and n are discrete.

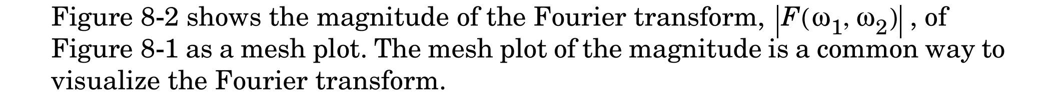


Figure 8-2 shows the magnitude of the Fourier transform, $|F(\omega_1, \omega_2)|$, of Figure 8-1 as a mesh plot. The mesh plot of the magnitude is a common way to visualize the Fourier transform.

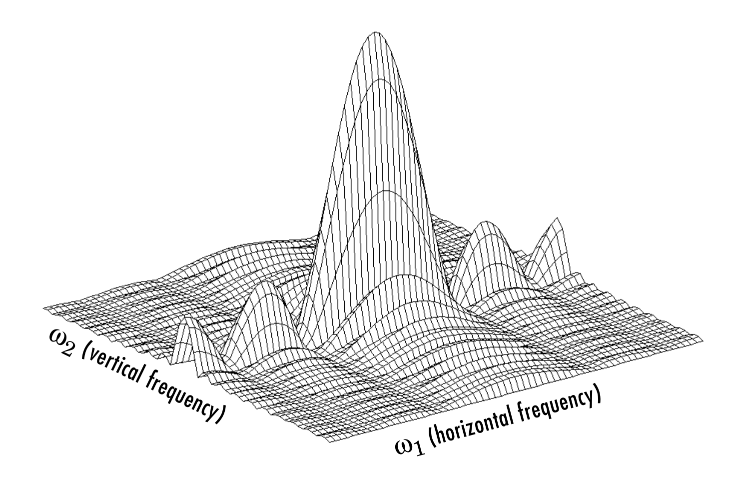


Figure 8-2: Magnitude Image of a Rectangular Function

The peak at the center of the plot is $F(0, 0)$, which is the sum of all the values in $f(m, n)$. The plot also shows that $F(\omega_1, \omega_2)$ has more energy at high horizontal frequencies than at high vertical frequencies. This reflects the fact that horizontal cross sections of $f(m, n)$ are narrow pulses, while vertical cross sections are broad pulses. Narrow pulses have more high-frequency content than broad pulses.

Another common way to visualize the Fourier transform is to display $\log|F(\omega_1, \omega_2)|$ as an image, as in

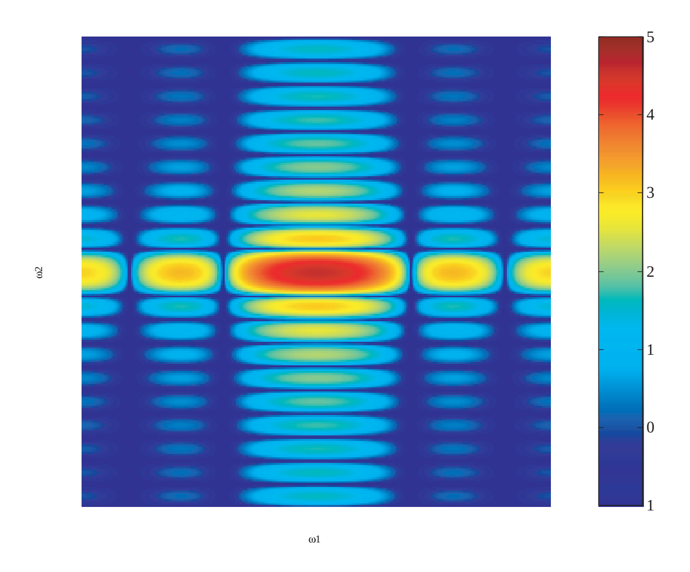


Figure 8-3: The Log of the Fourier Transform of a Rectangular Function

Using the logarithm helps to bring out details of the Fourier transform in regions where $F(\omega_1, \omega_2)$ is very close to 0.

Examples of the Fourier transform for other simple shapes are shown below.

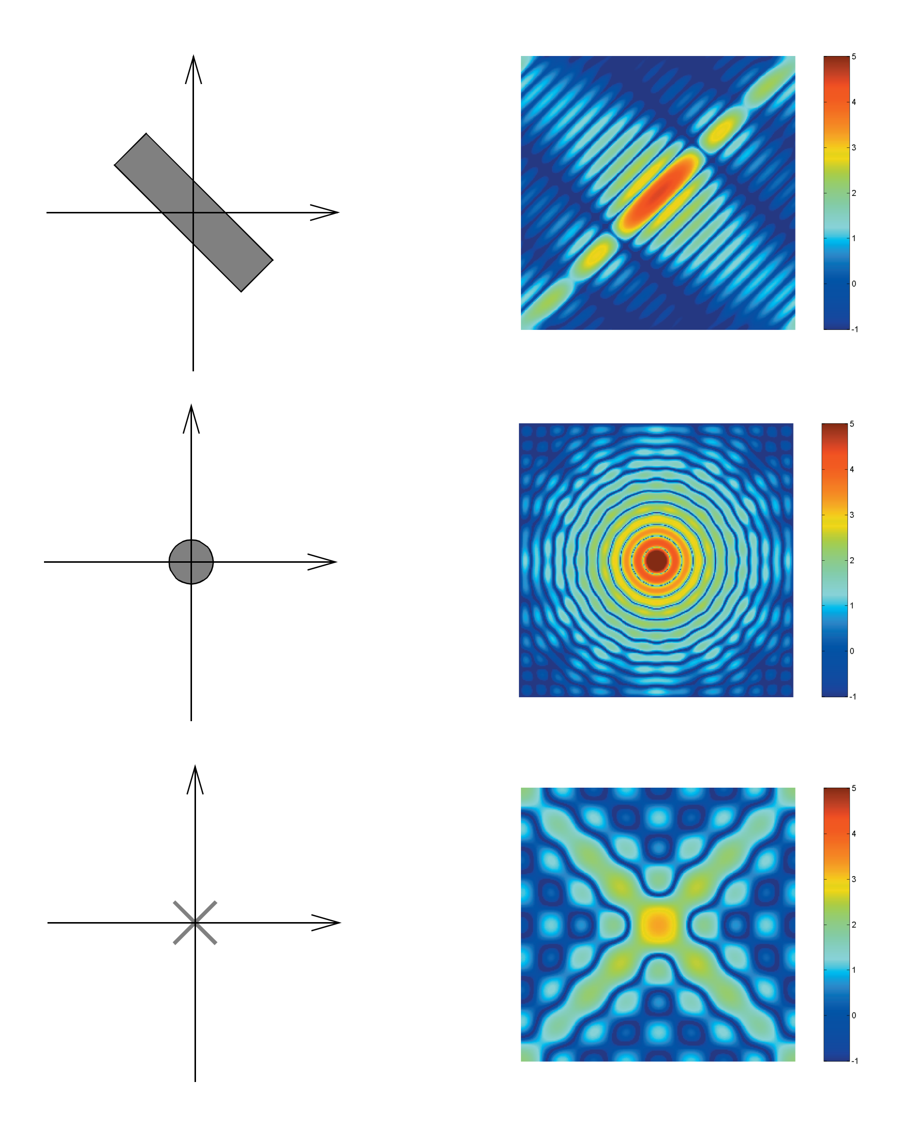


Figure 8-4: Fourier Transforms of Some Simple Shapes

Discrete Fourier Transform

Working with the Fourier transform on a computer usually involves a form of the transform known as the discrete Fourier transform (DFT). There are two principal reasons for using this form:

- The input and output of the DFT are both discrete, which makes it convenient for computer manipulations.
- There is a fast algorithm for computing the DFT known as the fast Fourier transform (FFT).

The DFT is usually defined for a discrete function $f(m, n)$ that is nonzero only over the finite region $0 \le m \le M-1$ and $0 \le n \le N-1$. The two-dimensional M-by-N DFT and inverse M-by-N DFT relationships are given by

$$F(p, q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) e^{-j(2\pi/M)pm} e^{-j(2\pi/N)qn} \qquad \begin{matrix} p = 0, 1, \ldots, M-1 \\ q = 0, 1, \ldots, N-1 \end{matrix}$$

$$f(m, n) = \frac{1}{MN} \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) e^{j(2\pi/M)pm} e^{j(2\pi/N)qn} \qquad \begin{matrix} m = 0, 1, \ldots, M-1 \\ n = 0, 1, \ldots, N-1 \end{matrix}$$

The values $F(p, q)$ are the DFT coefficients of $f(m, n)$. The zero-frequency coefficient, $F(0, 0)$ is often called the "DC component." DC is an electrical engineering term that stands for direct current. (Note that matrix indices in MATLAB always start at 1 rather than 0; therefore, the matrix elements `f(1,1)` and `F(1,1)` correspond to the mathematical quantities $f(0, 0)$ and $F(0, 0)$, respectively.)

The MATLAB functions `fft`, `fft2`, and `fftn` implement the fast Fourier transform algorithm for computing the one-dimensional DFT, two-dimensional DFT, and N-dimensional DFT, respectively. The functions `ifft`, `ifft2`, and `ifftn` compute the inverse DFT.

Relationship to the Fourier Transform

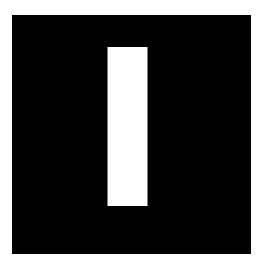
The DFT coefficients $F(p, q)$ are samples of the Fourier transform $F(\omega_1, \omega_2)$.

$$F(p, q) = F(\omega_1, \omega_2)\Big|_{\substack{\omega_1 = 2\pi p/M \\ \omega_2 = 2\pi q/N}} \qquad \begin{matrix} p = 0, 1, \ldots, M-1 \\ q = 0, 1, \ldots, N-1 \end{matrix}$$

Example

Let's construct a matrix `f` that is similar to the function *f(m,n)* in the example in "Definition of Fourier Transform" on page 8-4. Remember that *f(m,n)* is equal to 1 within the rectangular region and 0 elsewhere. We use a binary image to represent *f(m,n)*.

```
f = zeros(30,30);
f(5:24,13:17) = 1;
imshow(f,'notruesize')
```



Compute and visualize the 30-by-30 DFT of `f` with these commands.

```
F = fft2(f);
F2 = log(abs(F));
imshow(F2,[-1 5],'notruesize'); colormap(jet); colorbar
```

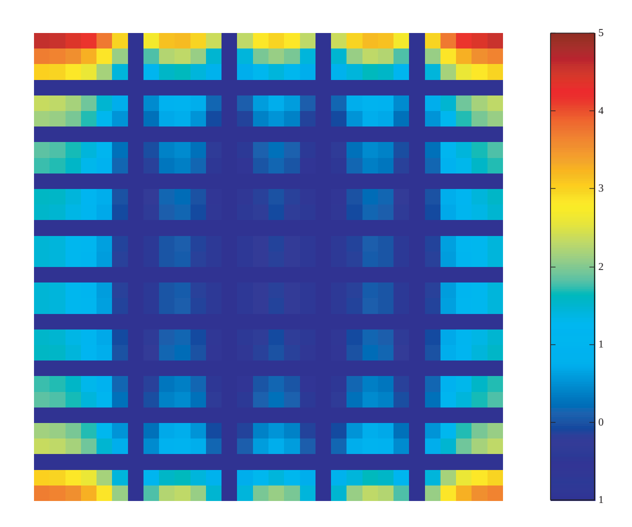


Figure 8-5: A Discrete Fourier Transform Computed Without Padding

This plot differs from the Fourier transform displayed on Figure 8-3. First, the sampling of the Fourier transform is much coarser. Second, the zero-frequency coefficient is displayed in the upper-left corner instead of the traditional location in the center.

We can obtain a finer sampling of the Fourier transform by zero-padding `f` when computing its DFT. The zero-padding and DFT computation can be performed in a single step with this command.

```
F = fft2(f,256,256);
```

This command zero-pads `f` to be 256-by-256 before computing the DFT.

```
imshow(log(abs(F)),[-1 5]); colormap(jet); colorbar
```

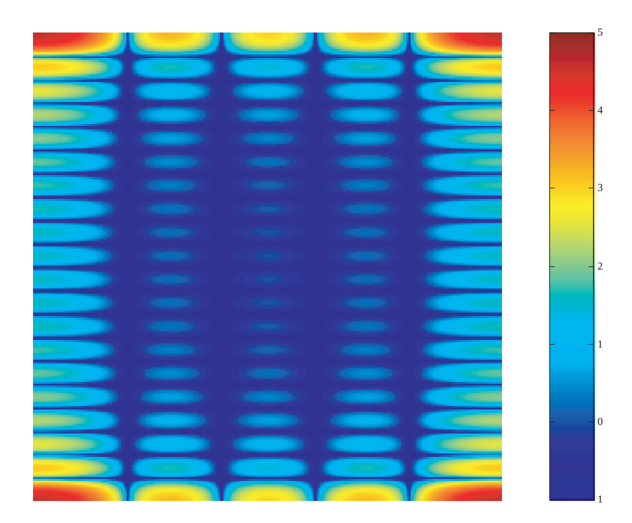


Figure 8-6: A Discrete Fourier Transform Computed with Padding

The zero-frequency coefficient, however, is still displayed in the upper-left corner rather than the center. You can fix this problem by using the function `fftshift`, which swaps the quadrants of `F` so that the zero-frequency coefficient is in the center.

```
F = fft2(f,256,256);
F2 = fftshift(F);
imshow(log(abs(F2)),[-1 5]); colormap(jet); colorbar
```

The resulting plot is identical to the one on Figure 8-3.

Applications

This section presents a few of the many image processing-related applications of the Fourier transform.

Frequency Response of Linear Filters

The Fourier transform of the impulse response of a linear filter gives the frequency response of the filter. The function `freqz2` computes and displays a filter's frequency response. The frequency response of the Gaussian

convolution kernel shows that this filter passes low frequencies and attenuates high frequencies.

```
h = fspecial('gaussian');
freqz2(h)
```

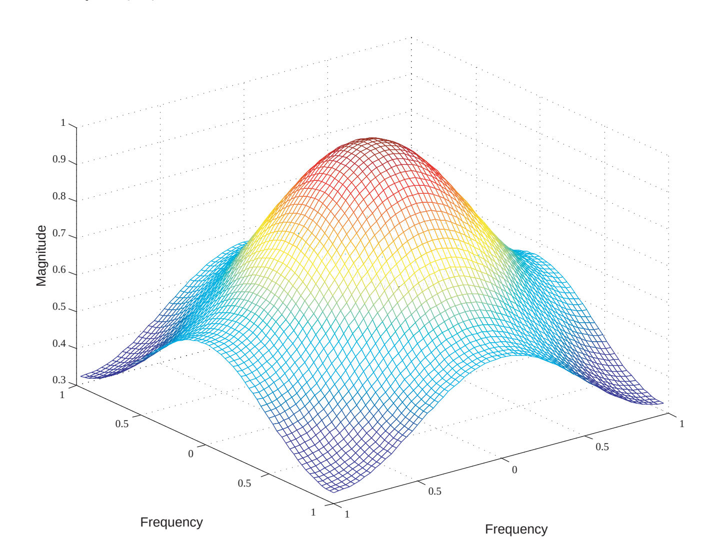


Figure 8-7: The Frequency Response of a Gaussian Filter

See “Linear Filtering and Filter Design” on page 7-1 for more information about linear filtering, filter design, and frequency responses.

Fast Convolution

A key property of the Fourier transform is that the multiplication of two Fourier transforms corresponds to the convolution of the associated spatial functions. This property, together with the fast Fourier transform, forms the basis for a fast convolution algorithm.

Suppose that `A` is an M-by-N matrix and `B` is a P-by-Q matrix. The convolution of `A` and `B` can be computed using the following steps:

1. Zero-pad `A` and `B` so that they are at least (M+P–1)-by-(N+Q–1). (Often `A` and `B` are zero-padded to a size that is a power of 2 because `fft2` is fastest for these sizes.)

2 Compute the two-dimensional DFT of `A` and `B` using `fft2`.

3 Multiply the two DFTs together.

4 Using `ifft2`, compute the inverse two-dimensional DFT of the result from step 3.

For example,

```
A = magic(3);
B = ones(3);
A(8,8) = 0;                        % Zero pad A to be 8 by 8
B(8,8) = 0;                        % Zero pad B to be 8 by 8
C = ifft2(fft2(A).*fft2(B));
C = C(1:5,1:5);                    % Extract the nonzero portion
C = real(C)    % Remove imaginary part caused by roundoff error

C =

    8.0000    9.0000   15.0000    7.0000    6.0000
   11.0000   17.0000   30.0000   19.0000   13.0000
   15.0000   30.0000   45.0000   30.0000   15.0000
    7.0000   21.0000   30.0000   23.0000    9.0000
    4.0000   13.0000   15.0000   11.0000    2.0000
```

The FFT-based convolution method is most often used for large inputs. For small inputs it is generally faster to use `imfilter`.

Locating Image Features

The Fourier transform can also be used to perform correlation, which is closely related to convolution. Correlation can be used to locate features within an image; in this context correlation is often called *template matching*.

For instance, suppose you want to locate occurrences of the letter “a” in an image containing text. This example reads in `text.tif` and creates a template image by extracting a letter “a” from it.

```
bw = imread('text.tif');
a=bw(59:71,81:91); %Extract one of the letters  a  from the image.
imshow(bw);
figure, imshow(a);
```

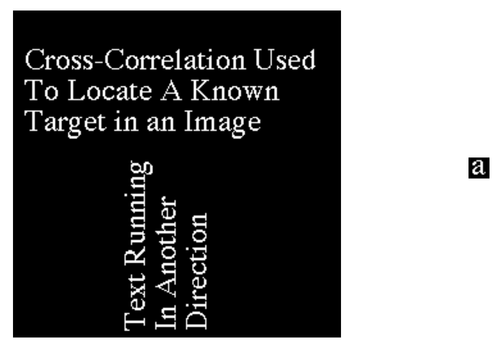


Figure 8-8: An Image (left) and the Template to Correlate (right)

The correlation of the image of the letter “a” with the larger image can be computed by first rotating the image of “a” by 180° and then using the FFT-based convolution technique described above. (Note that convolution is equivalent to correlation if you rotate the convolution kernel by 180°.) To match the template to the image, you can use the `fft2` and `ifft2` functions.

```
C = real(ifft2(fft2(bw) .* fft2(rot90(a,2),256,256)));
figure, imshow(C,[])%Display, scaling data to appropriate range.
max(C(:)) %Find max pixel value in C.

ans =

   51.0000

thresh = 45; %Use a threshold that s a little less than max.
figure, imshow(C > thresh)%Display showing pixels over threshold.
```



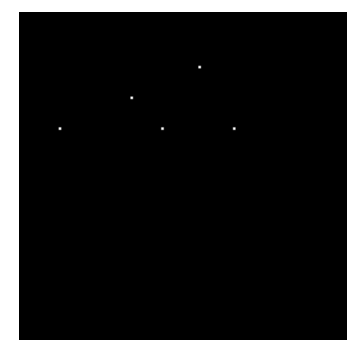


Figure 8-9: A Correlated Image (left) and its Thresholded Result (right)

The left image above is the result of the correlation; bright peaks correspond to occurrences of the letter. The locations of these peaks are indicated by the white spots in the thresholded correlation image shown on the right.

Note that you could also have created the template image by zooming in on the image and using the interactive version of `imcrop`. For example, with `text.tif` displayed in the current figure window, enter

```
zoom on
a = imcrop
```

To determine the coordinates of features in an image, you can use the `pixval` function.

Discrete Cosine Transform

The discrete cosine transform (DCT) represents an image as a sum of sinusoids of varying magnitudes and frequencies. The `dct2` function in the Image Processing Toolbox computes the two-dimensional discrete cosine transform (DCT) of an image. The DCT has the property that, for a typical image, most of the visually significant information about the image is concentrated in just a few coefficients of the DCT. For this reason, the DCT is often used in image compression applications. For example, the DCT is at the heart of the international standard lossy image compression algorithm known as JPEG. (The name comes from the working group that developed the standard: the Joint Photographic Experts Group.)

The two-dimensional DCT of an M-by-N matrix `A` is defined as follows.

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{matrix}$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases} \qquad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

The values B_{pq} are called the *DCT coefficients* of `A`. (Note that matrix indices in MATLAB always start at 1 rather than 0; therefore, the MATLAB matrix elements `A(1,1)` and `B(1,1)` correspond to the mathematical quantities A_{00} and B_{00}, respectively.)

The DCT is an invertible transform, and its inverse is given by

$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le m \le M-1 \\ 0 \le n \le N-1 \end{matrix}$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases} \qquad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

The inverse DCT equation can be interpreted as meaning that any M-by-N matrix `A` can be written as a sum of MN functions of the form

$$\alpha_p \alpha_q \cos\frac{\pi(2m+1)p}{2M}\cos\frac{\pi(2n+1)q}{2N}, \qquad \begin{matrix} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{matrix}$$

These functions are called the *basis functions* of the DCT. The DCT coefficients B_{pq}, then, can be regarded as the *weights* applied to each basis function. For 8-by-8 matrices, the 64 basis functions are illustrated by this image.

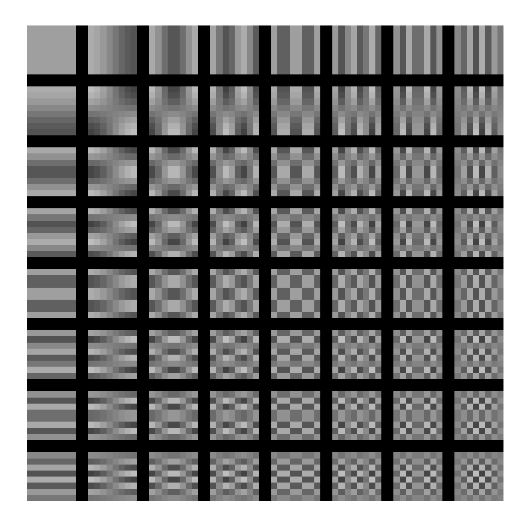


Figure 8-10: The 64 Basis Functions of an 8-by-8 Matrix

Horizontal frequencies increase from left to right, and vertical frequencies increase from top to bottom. The constant-valued basis function at the upper left is often called the *DC basis function*, and the corresponding DCT coefficient B_{00} is often called the *DC coefficient*.

The DCT Transform Matrix

The Image Processing Toolbox offers two different ways to compute the DCT. The first method is to use the function `dct2`. `dct2` uses an FFT-based algorithm for speedy computation with large inputs. The second method is to use the DCT *transform matrix*, which is returned by the function `dctmtx` and may be more efficient for small square inputs, such as 8-by-8 or 16-by-16. The M-by-M transform matrix `T` is given by

$$T_{pq} = \begin{cases} \frac{1}{\sqrt{M}} & p = 0, \quad 0 \le q \le M-1 \\ \sqrt{\frac{2}{M}} \cos \frac{\pi(2q+1)p}{2M} & 1 \le p \le M-1, \quad 0 \le q \le M-1 \end{cases}$$

For an M-by-M matrix `A`, `T*A` is an M-by-M matrix whose columns contain the one-dimensional DCT of the columns of `A`. The two-dimensional DCT of `A` can be computed as `B=T*A*T'`. Since `T` is a real orthonormal matrix, its inverse is the same as its transpose. Therefore, the inverse two-dimensional DCT of `B` is given by `T'*B*T`.

The DCT and Image Compression

In the JPEG image compression algorithm, the input image is divided into 8-by-8 or 16-by-16 blocks, and the two-dimensional DCT is computed for each block. The DCT coefficients are then quantized, coded, and transmitted. The JPEG receiver (or JPEG file reader) decodes the quantized DCT coefficients, computes the inverse two-dimensional DCT of each block, and then puts the blocks back together into a single image. For typical images, many of the DCT coefficients have values close to zero; these coefficients can be discarded without seriously affecting the quality of the reconstructed image.

The example code below computes the two-dimensional DCT of 8-by-8 blocks in the input image; discards (sets to zero) all but 10 of the 64 DCT coefficients in each block; and then reconstructs the image using the two-dimensional inverse DCT of each block. The transform matrix computation method is used.

```
I = imread('cameraman.tif');
I = im2double(I);
T = dctmtx(8);
B = blkproc(I,[8 8],'P1*x*P2',T,T');
mask = [1   1   1   1   0   0   0   0
        1   1   1   0   0   0   0   0
        1   1   0   0   0   0   0   0
        1   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0];
```

```
B2 = blkproc(B,[8 8],'P1.*x',mask);
I2 = blkproc(B2,[8 8],'P1*x*P2',T',T);
imshow(I), figure, imshow(I2)
```





Although there is some loss of quality in the reconstructed image, it is clearly recognizable, even though almost 85% of the DCT coefficients were discarded. To experiment with discarding more or fewer coefficients, and to apply this technique to other images, try running the demo function `dctdemo`.

Radon Transform

The radon transform represents an image as a collection of projections along various directions. It is used in areas ranging from seismology to computer vision

The `radon` function in the Image Processing Toolbox computes *projections* of an image matrix along specified directions. A projection of a two-dimensional function $f(x,y)$ is a line integral in a certain direction. For example, the line integral of $f(x,y)$ in the vertical direction is the projection of $f(x,y)$ onto the x-axis; the line integral in the horizontal direction is the projection of $f(x,y)$ onto the y-axis. Figure 8-11 shows horizontal and vertical projections for a simple two-dimensional function.

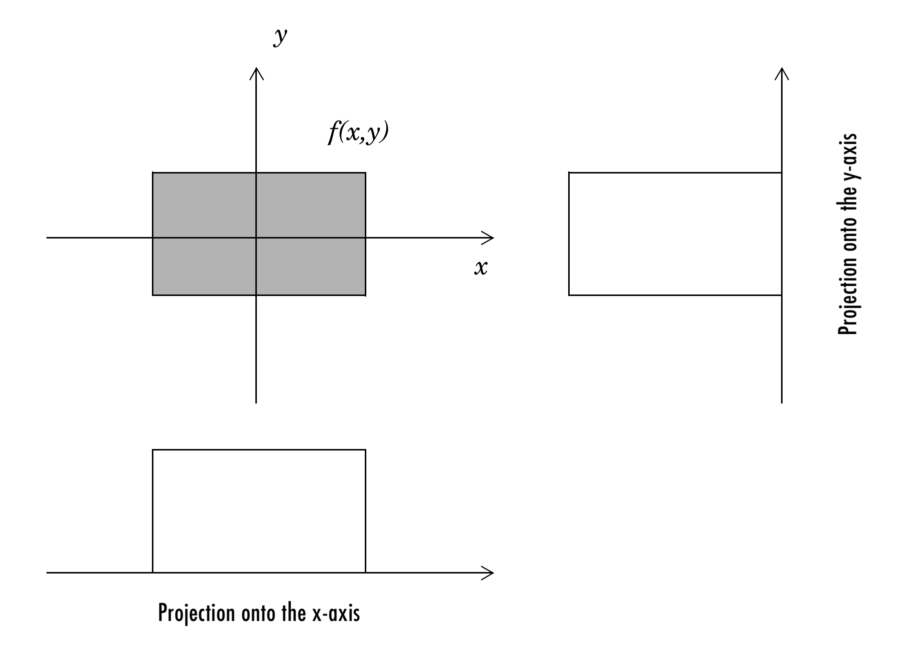


Figure 8-11: Horizontal and Vertical Projections of a Simple Function

Projections can be computed along any angle θ. In general, the Radon transform of $f(x,y)$ is the line integral of f parallel to the y' axis

$$R_\theta(x') = \int_{-\infty}^{\infty} f(x'\cos\theta - y'\sin\theta,\ x'\sin\theta + y'\cos\theta)\ dy'$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The figure below illustrates the geometry of the Radon transform.

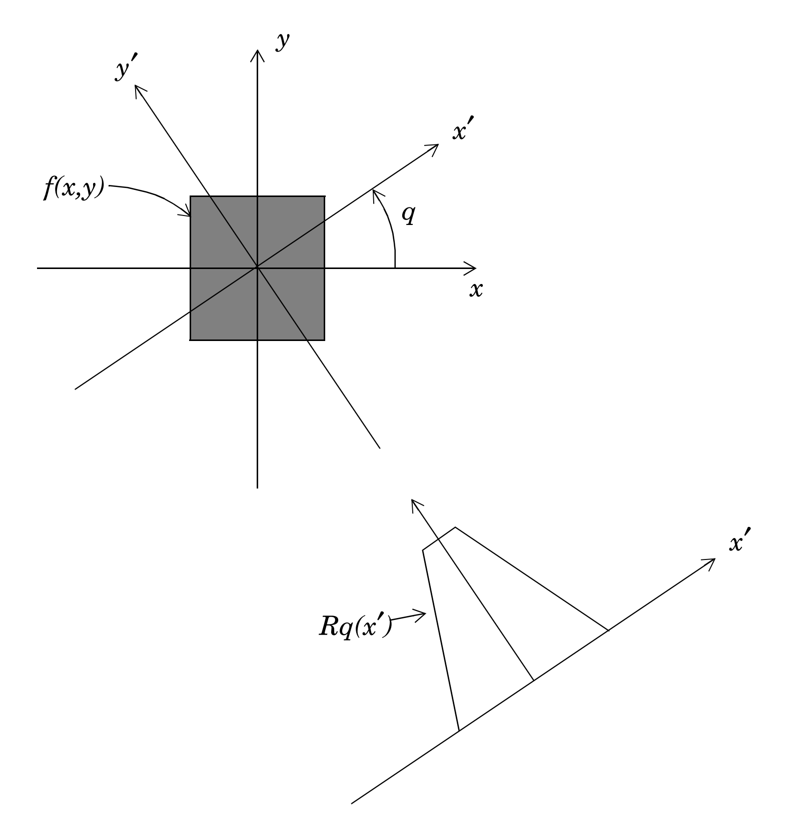


Figure 8-12: The Geometry of the Radon Transform

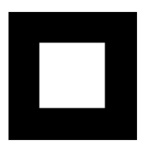
This command computes the Radon transform of `I` for the angles specified in the vector `theta`

```
[R,xp] = radon(I,theta);
```

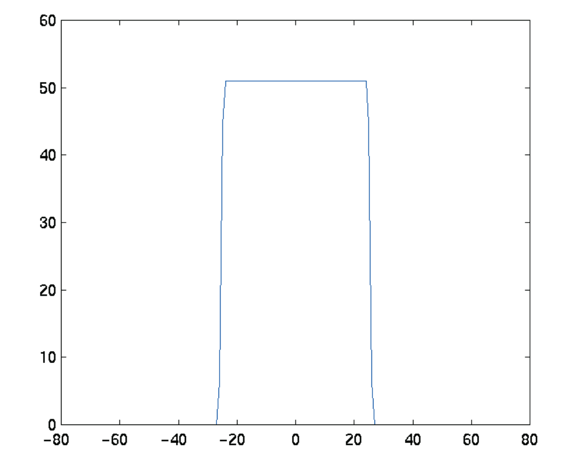
The columns of R contain the Radon transform for each angle in theta. The vector xp contains the corresponding coordinates along the x'-axis. The "center pixel" of I is defined to be floor((size(I)+1)/2); this is the pixel on the x'-axis corresponding to $x' = 0$.

The commands below compute and plot the Radon transform at 0° and 45° of an image containing a single square object.

```
I = zeros(100,100);
I(25:75, 25:75) = 1;
imshow(I)
```



```
[R,xp] = radon(I,[0 45]);
figure; plot(xp,R(:,1)); title('R_{0^o} (x\prime)')
figure; plot(xp,R(:,2)); title('R_{45^o} (x\prime)')
```



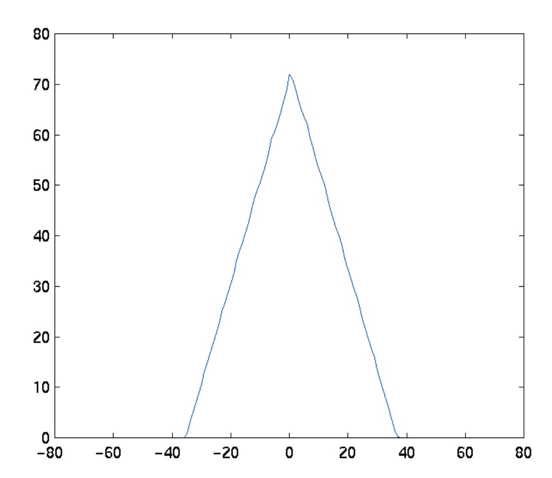


Figure 8-13: Two Radon Transforms of a Square Function

Note `xp` is the same for all projection angles.

The Radon transform for a large number of angles is often displayed as an image. In this example, the Radon transform for the square image is computed at angles from 0° to 180°, in 1° increments.

```
theta = 0:180;
[R,xp] = radon(I,theta);
imagesc(theta,xp,R);
title('R_{\theta} (X\prime)');
xlabel('\theta (degrees)');
ylabel('X\prime');
set(gca,'XTick',0:20:180);
colormap(hot);
colorbar
```

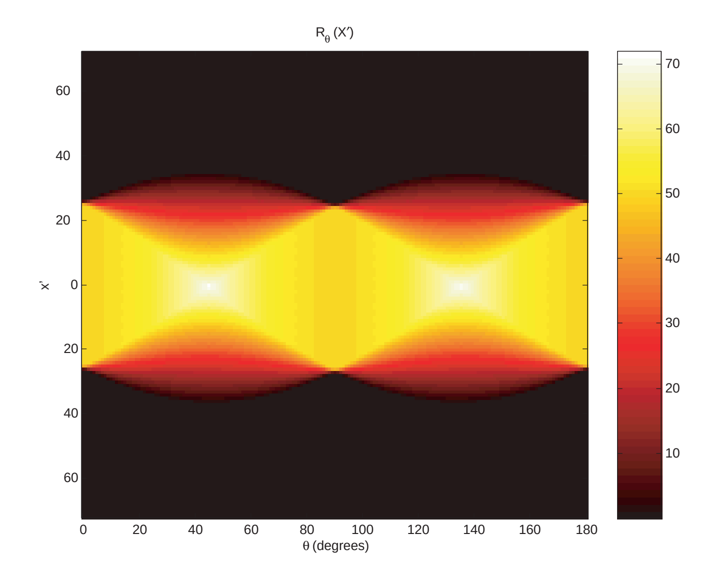


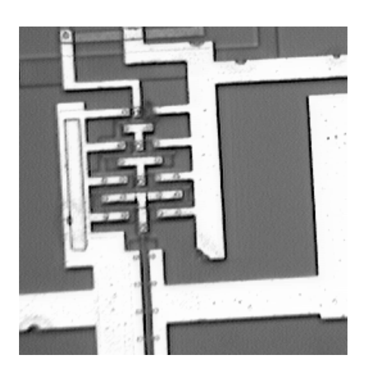
Figure 8-14: A Radon Transform Using 180 Projections

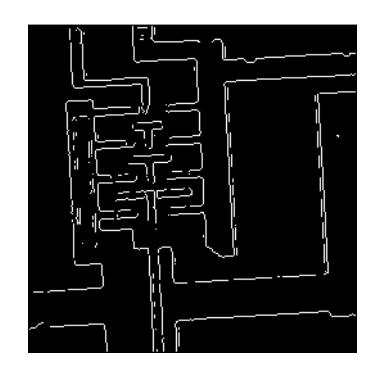
Using the Radon Transform to Detect Lines

The Radon transform is closely related to a common computer vision operation known as the Hough transform. You can use the radon function to implement a form of the Hough transform used to detect straight lines. The steps are:

1 Compute a binary edge image using the edge function.

```
I = imread('ic.tif');
BW = edge(I);
imshow(I)
figure, imshow(BW)
```





2 Compute the Radon transform of the edge image.

```
theta = 0:179;
[R,xp] = radon(BW,theta);
figure, imagesc(theta, xp, R); colormap(hot);
xlabel('\theta (degrees)'); ylabel('X\prime');
title('R_{\theta} (X\prime)');
colorbar
```

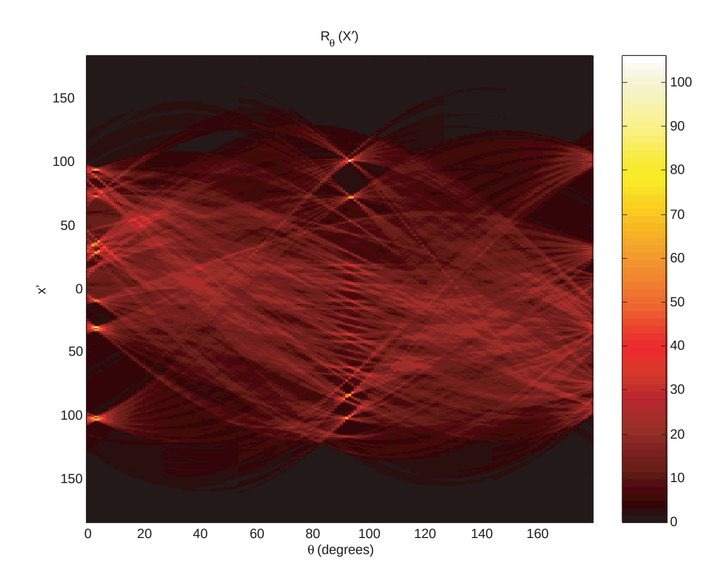


Figure 8-15: Radon Transform of an Edge Image

3 Find the locations of strong peaks in the Radon transform matrix. The locations of these peaks correspond to the location of straight lines in the original image.

In this example, the strongest peak in `R` corresponds to $\theta = 94°$ and $x' = -101$. The line perpendicular to that angle and located at $x' = -101$ is shown below, superimposed in red on the original image. The Radon transform geometry is shown in black.

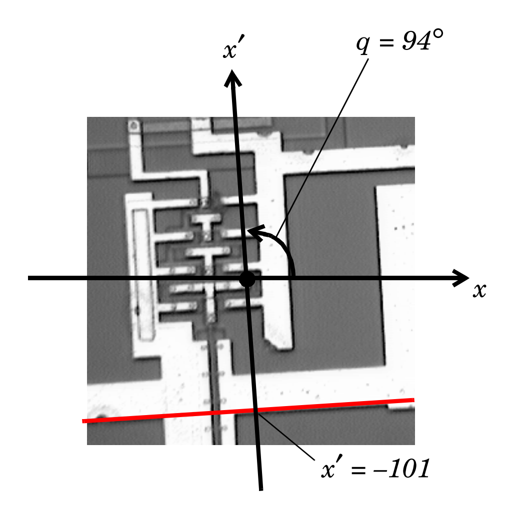


Figure 8-16: The Radon Transform Geometry and the Strongest Peak (Red)

Notice that the other strong lines parallel to the red line also appear as peaks at $\theta = 94°$ in the transform. Also, the lines perpendicular to this line appear as peaks at $\theta = 4°$.

The Inverse Radon Transform

The `iradon` function performs the inverse Radon transform, which is commonly used in tomography applications. This transform inverts the Radon transform (which was introduced in the previous section), and can therefore be used to reconstruct images from projection data.

As discussed in the previous section “Radon Transform” on page 8-21, given an image `I` and a set of angles `theta`, the function `radon` can be used to calculate the Radon transform.

```
R = radon(I,theta);
```

The function `iradon` can then be called to reconstruct the image `I`.

```
IR = iradon(R,theta);
```

In the example above, projections are calculated from the original image `I`. In most application areas, there is no *original image* from which projections are

formed. For example, in X-ray absorption tomography, projections are formed by measuring the attenuation of radiation that passes through a physical specimen at different angles. The original image can be thought of as a cross section through the specimen, in which intensity values represent the density of the specimen. Projections are collected using special purpose hardware, and then an internal image of the specimen is reconstructed by `iradon`. This allows for noninvasive imaging of the inside of a living body or another opaque object.

`iradon` reconstructs an image from parallel beam projections. In *parallel beam geometry*, each projection is formed by combining a set of line integrals through an image at a specific angle.

Figure 8-17 below illustrates how parallel beam geometry is applied in X-ray absorption tomography. Note that there is an equal number of n emitters and n detectors. Each detector measures the radiation emitted from its corresponding emitter, and the attenuation in the radiation gives a measure of the integrated density, or mass, of the object. This corresponds to the line integral that is calculated in the Radon transform.

The parallel beam geometry used in the figure is the same as the geometry that was described under "Radon Transform" on page 8-21. `f(x,y)` denotes the brightness of the image and $Rq(x')$ is the projection at angle q.

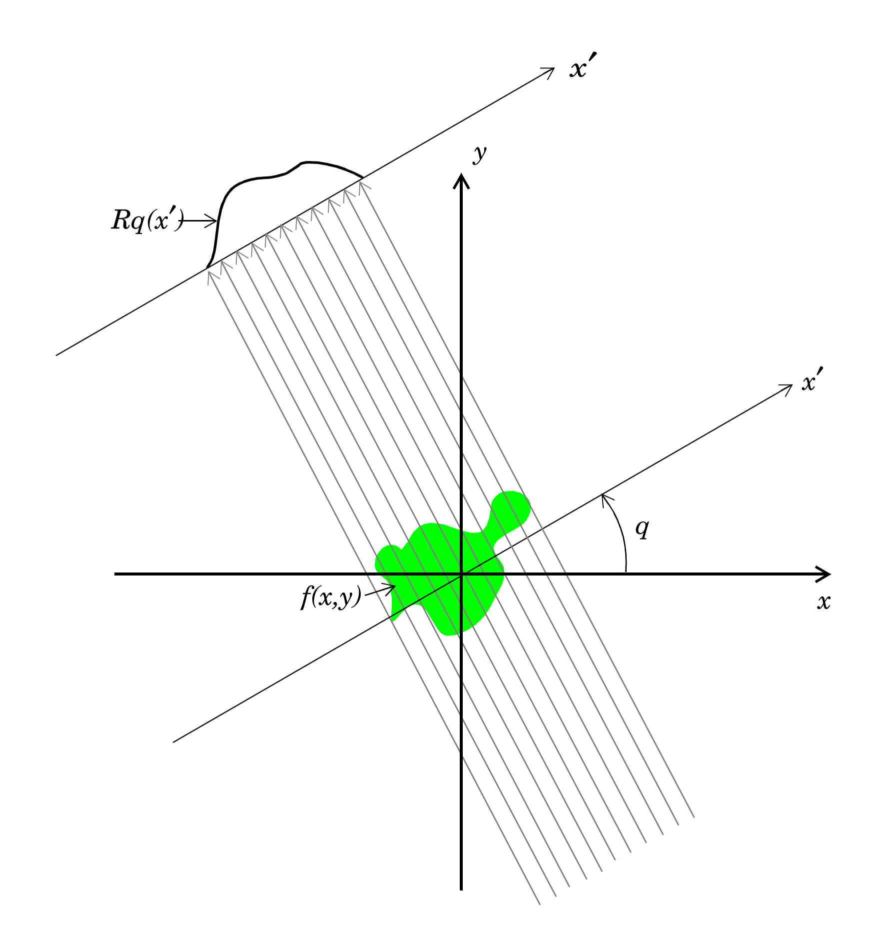


Figure 8-17: Parallel Beam Projections Through an Object

Another geometry that is commonly used is *fan beam* geometry, in which there is one emitter and *n* detectors. There are methods for resorting sets of fan beam projections into parallel beam projections, which can then be used by `iradon`. (For more information on these methods, see Kak & Slaney, *Principles of Computerized Tomographic Imaging*, IEEE Press, NY, 1988, pp. 92-93.)

`iradon` uses the *filtered backprojection* algorithm to compute the inverse Radon transform. This algorithm forms an approximation to the image `I` based on the projections in the columns of `R`. A more accurate result can be obtained by using more projections in the reconstruction. As the number of projections (the length

of theta) increases, the reconstructed image IR more accurately approximates the original image I. The vector theta must contain monotonically increasing angular values with a constant incremental angle $\Delta\theta$. When the scalar $\Delta\theta$ is known, it can be passed to iradon instead of the array of theta values. Here is an example.

```
IR = iradon(R,Dtheta);
```

The filtered backprojection algorithm filters the projections in R and then reconstructs the image using the filtered projections. In some cases, noise can be present in the projections. To remove high frequency noise, apply a window to the filter to attenuate the noise. Many such windowed filters are available in iradon. The example call to iradon below applies a Hamming window to the filter. See the iradon reference page for more information.

```
IR = iradon(R,theta,'Hamming');
```

iradon also enables you to specify a normalized frequency, D, above which the filter has zero response. D must be a scalar in the range [0,1]. With this option, the frequency axis is rescaled, so that the whole filter is compressed to fit into the frequency range [0,D]. This can be useful in cases where the projections contain little high frequency information but there is high frequency noise. In this case, the noise can be completely suppressed without compromising the reconstruction. The following call to iradon sets a normalized frequency value of 0.85.

```
IR = iradon(R,theta,0.85);
```

Examples

The commands below illustrate how to use radon and iradon to form projections from a sample image and then reconstruct the image from the projections. The test image is the Shepp-Logan head phantom, which can be generated by the Image Processing Toolbox function phantom. The phantom image illustrates many of the qualities that are found in real-world tomographic imaging of human heads. The bright elliptical shell along the exterior is analogous to a skull, and the many ellipses inside are analogous to brain features or tumors.

```
P = phantom(256);
imshow(P)
```



As a first step the Radon transform of the phantom brain is calculated for three different sets of theta values. R1 has 18 projections, R2 has 36 projections, and R3 had 90 projections.

```
theta1 = 0:10:170; [R1,xp] = radon(P,theta1);
theta2 = 0:5:175;  [R2,xp] = radon(P,theta2);
theta3 = 0:2:178;  [R3,xp] = radon(P,theta3);
```

Now the Radon transform of the Shepp-Logan Head phantom is displayed using 90 projections (R3).

```
figure, imagesc(theta3,xp,R3); colormap(hot); colorbar
xlabel('\theta'); ylabel('x\prime');
```

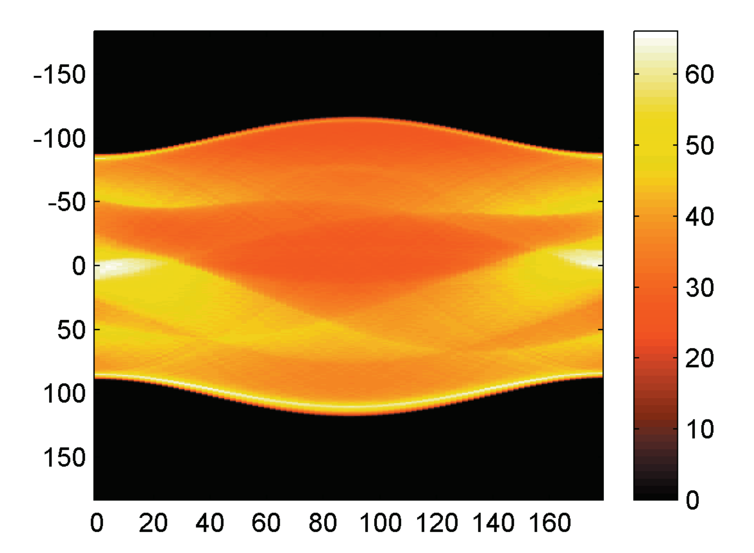


Figure 8-18: Radon Transform of Head Phantom Using 90 Projections

When we look at Figure 8-18, we can see some of the features of the input image. The first column in the Radon transform corresponds to a projection at 0° which is integrating in the vertical direction. The centermost column corresponds to a projection at 90°, which is integrating in the horizontal direction. The projection at 90° has a wider profile than the projection at 0° due to the larger vertical semi-axis of the outermost ellipse of the phantom.

Figure 8-19 shows the inverse Radon transforms of `R1`, `R2`, and `R3`, which were generated above. Image `I1` was reconstructed with the projections in `R1`, and it is the least accurate reconstruction, because it has the fewest projections. `I2` was reconstructed with the 36 projections in `R2`, and the quality of the reconstruction is better, but it is still not clear enough to discern clearly the three small ellipses in the lower portion of the test image. `I3` was reconstructed using the 90 projections in `R3`, and the result closely resembles the original image. Notice that when the number of projections is relatively small (as in `I1` and `I2`), the reconstruction may include some artifacts from the back projection. To avoid this, use a larger number of angles.

```
I1 = iradon(R1,10);
I2 = iradon(R2,5);
I3 = iradon(R3,2);
imshow(I1)
figure, imshow(I2)
figure, imshow(I3)
```

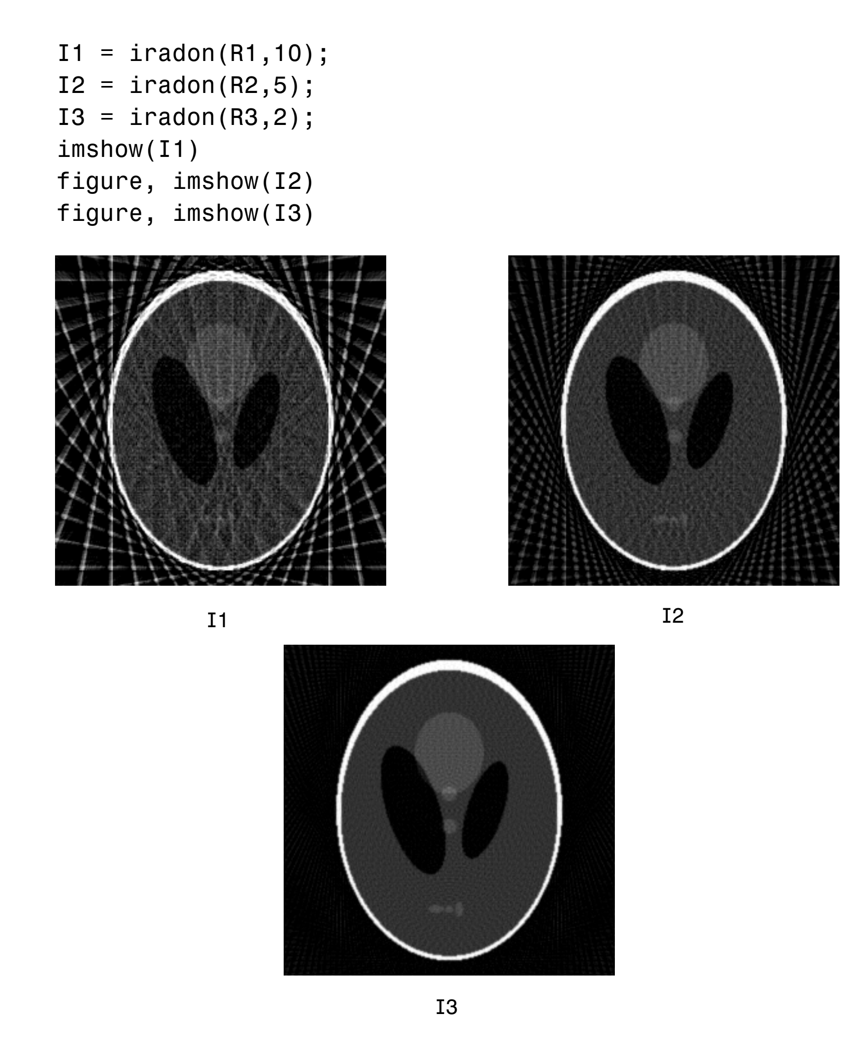


Figure 8-19: Inverse Radon Transforms of the Shepp-Logan Head Phantom

9

Morphological Operations

Morphology is a technique of image processing based on shapes. The value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. By choosing the size and shape of the neighborhood, you can construct a morphological operation that is sensitive to specific shapes in the input image.

This section describes the Image Processing toolbox morphological functions. You can use these functions to perform common image processing tasks, such as contrast enhancement, noise removal, thinning, skeletonization, filling, and segmentation. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Background	In a binary image, pixels that are off, i.e., set to the value 0, are considered the background. When you view a binary image, the background pixels appear black.
Connectivity	The criteria that describes how pixels in an image form a connected group. For example, a connected component is “8-connected” if diagonally adjacent pixels are considered to be touching, otherwise, it is “4-connected.” The toolbox supports 2-D as well as multidimensional connectivities. See “Pixel Connectivity” on page 9-23 for more information.
Foreground	In a binary image, pixels that are on, i.e., set to the value 1, are considered the foreground. When you view a binary image, the foreground pixels appear white.
Global maxima	The highest regional maxima in the image. See the entry for regional maxima in this table for more information.
Global minima	The lowest regional minima in the image. See the entry for regional minima in this table for more information.
Morphology	A broad set of image processing operations that process images based on shapes. Morphological operations apply a structuring element to an input image, creating an output image of the same size. The most basic morphological operations are dilation and erosion.
Neighborhood	A set of pixels that are defined by their locations relative to the pixel of interest. A neighborhood can be defined by a structuring element or by specifying a connectivity.
Object	A set of pixels in a binary image that form a connected group. In the context of this chapter, “object” and “connected component” are equivalent.

Terms	Definitions
Packed binary image	A method of compressing binary images that can speed up the processing of the image.
Regional maxima	A connected set of pixels of constant intensity from which it is impossible to reach a point with higher intensity without first descending; that is, a connected component of pixels with the same intensity value, t, surrounded by pixels that all have a value less than t.
Regional minima	A connected set of constant intensity from which it is impossible to reach a point with lower intensity without first ascending; that is, a connected component of pixels with the same intensity value, t, surrounded by pixels that all have a value greater than t.
Structuring element	A matrix used to define a neighborhood shape and size for morphological operations, including dilation and erosion. It consists of only 0's and 1's and can have an arbitrary shape and size. The pixels with values of 1 define the neighborhood.

Dilation and Erosion

Dilation and erosion are two fundamental morphological operations. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries. The number of pixels added or removed from the objects in an image depends on the size and shape of the *structuring element* used to process the image.

The following sections describe the dilation and erosion functions in the toolbox:

- "Understanding Dilation and Erosion" — This section provides important background information about how the dilation and erosion functions operate.
- "Structuring Elements" on page 9-7 — This section describes structuring elements and how to create them.
- "Dilating an Image" on page 9-11 — This section describes how to use the `imdilate` function.
- "Eroding an Image" on page 9-12 — This section describes how to use the `imerode` function.
- "Combining Dilation and Erosion" on page 9-14 — This section describes some of the common operations that are based on dilation and erosion.
- "Dilation- and Erosion-Based Functions" on page 9-16 — This section describes toolbox functions that are based on dilation and erosion

Understanding Dilation and Erosion

In the morphological dilation and erosion operations, the state of any given pixel in the output image is determined by applying a rule to the corresponding pixel and its neighbors in the input image. The rule used to process the pixels defines the operation as a dilation or an erosion. This table lists the rules for both dilation and erosion.

Table 9-1: Rules for Grayscale Dilation and Erosion

Operation	Rule
Dilation	The value of the output pixel is the *maximum* value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to the value 1, the output pixel is set to 1.
Erosion	The value of the output pixel is the *minimum* value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to 0, the output pixel is set to 0.

The following figure illustrates the dilation of a binary image. In the figure, note how the structuring element defines the neighborhood of the pixel of interest, which is circled. (See "Structuring Elements" on page 9-7 for more information.) The dilation function applies the appropriate rule to the pixels in the neighborhood and assigns a value to the corresponding pixel in the output image. In the figure, the morphological dilation function sets the value of the output pixel to 1 because one of the elements in the neighborhood defined by the structuring element is on.

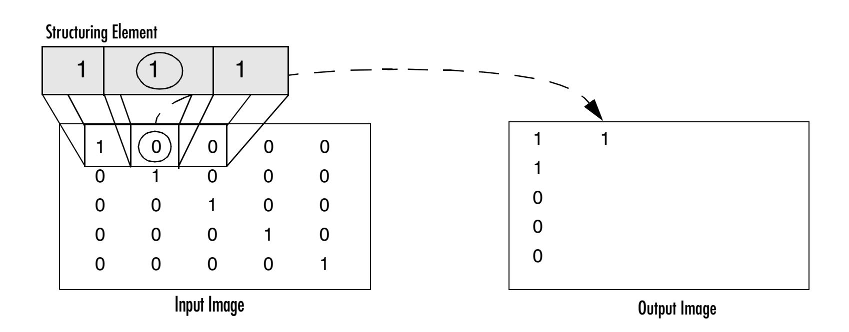


Figure 9-1: Morphological Processing of a Binary Image

The following figure illustrates this processing for a grayscale image. The figure shows the processing of a particular pixel in the input image. Note how the function applies the rule to the input pixel's neighborhood and uses the highest value of all the pixels in the neighborhood as the value of the corresponding pixel in the output image.

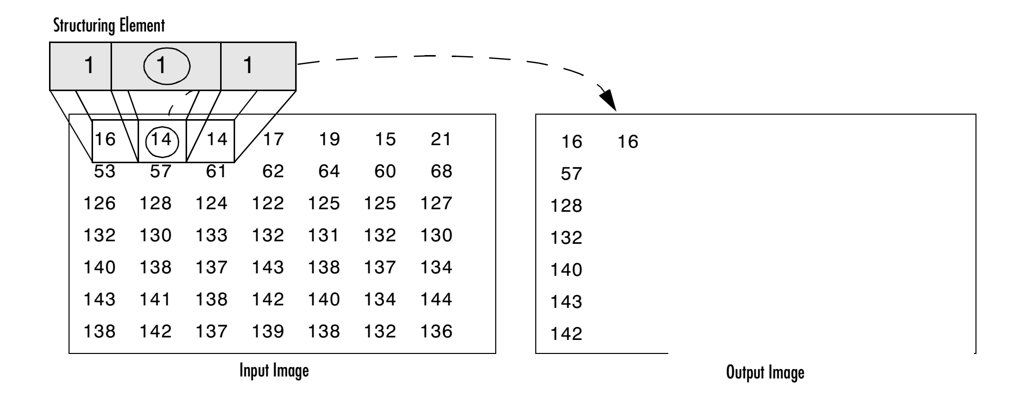


Figure 9-2: Morphological Processing of Grayscale Images

Processing Pixels at Image Borders

Morphological functions position the origin of the structuring element, its center element, over the pixel of interest in the input image. For pixels at the edge of an image, parts of the neighborhood defined by the structuring element can extend past the border of the image.

To process border pixels, the morphological functions assign a value to these undefined pixels, as if the functions had padded the image with additional rows and columns. The value of these "padding" pixels varies for dilation and erosion operations. The following table details the padding rules for dilation and erosion for both binary and grayscale images.

Table 9-2: Rules for Padding Images

Operation	Rule
Dilation	Pixels beyond the image border are assigned the *minimum* value afforded by the data type. For binary images, these pixels are assumed to be set to `0`. For grayscale images, the minimum value for `uint8` images is `0`.
Erosion	Pixels beyond the image border are assigned the *maximum* value afforded by the data type. For binary images, these pixels are assumed to be set to `1`. For grayscale images, the maximum value for `uint8` images is `255`.

Note By using the minimum value for dilation operations and the maximum value for erosion operations, the toolbox avoids *border effects*, where regions near the borders of the output image do not appear to be homogeneous with the rest of the image. For example, if erosion padded with a minimum value, eroding an image would result in a black border around the edge of the output image.

Structuring Elements

An essential part of the dilation and erosion operations is the structuring element used to probe the input image. Two-dimensional, or *flat*, structuring elements consist of a matrix of 0's and 1's, typically much smaller than the image being processed. The center pixel of the structuring element, called the *origin*, identifies the pixel of interest—the pixel being processed. The pixels in the structuring element containing 1's define the *neighborhood* of the structuring element. These pixels are also considered in the dilation or erosion processing. Three dimensional, or *nonflat*, structuring elements use 0's and 1's to define the extent of the structuring element in the x- and y-plane and add height values to define the third dimension.

The Origin of a Structuring Element

The morphological functions use this code to get the coordinates of the origin of structuring elements of any size and dimension.

```
origin = floor((size(nhood)+1)/2)
```

(In this code, `nhood` is the neighborhood defining the structuring element. Because structuring elements are MATLAB objects, you cannot use the size of the `STREL` object itself in this calculation. You must use the `STREL` `getnhood` method to retrieve the neighborhood of the structuring element from the `STREL` object. For information about other `STREL` object methods, see the `strel` function reference page.)

For example, the following illustrates a diamond-shaped structuring element.

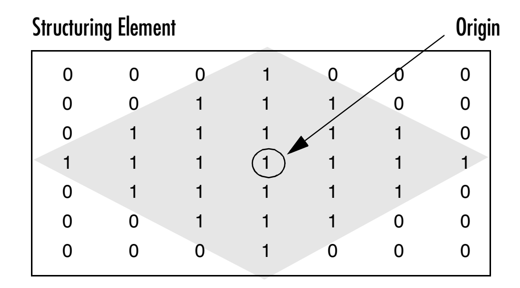


Figure 9-3: Origin of a Diamond-Shaped Structuring Element

Creating a Structuring Element

The toolbox dilation and erosion functions accept structuring element objects, called a `STREL`. You use the `strel` function to create `STREL`s of any arbitrary size and shape. The `strel` function also includes built-in support for many common shapes, such as lines, diamonds, disks, periodic lines, and balls.

Note You typically choose a structuring element the same size and shape as the objects you want to process in the input image. For example, to find lines in an image, create a linear structuring element.

For example, this code creates a flat, diamond-shaped structuring element.

```
se = strel('diamond',3)
se =

Flat STREL object containing 25 neighbors.
Decomposition: 3 STREL objects containing a total of 13 neighbors

Neighborhood:
     0     0     0     1     0     0     0
     0     0     1     1     1     0     0
     0     1     1     1     1     1     0
     1     1     1     1     1     1     1
     0     1     1     1     1     1     0
     0     0     1     1     1     0     0
     0     0     0     1     0     0     0
```

Structuring Element Decomposition

To enhance performance, the `strel` function may break structuring elements into smaller pieces, a technique known as *structuring element decomposition*.

For example, dilation by an 11-by-11 square structuring element can be accomplished by dilating first with a 1-by-11 structuring element, and then with an 11-by-1 structuring element. This results in a theoretical speed improvement of a factor of 5.5, although in practice the actual speed improvement is somewhat less.

Structuring element decompositions used for the `'disk'` and `'ball'` shapes are approximations; all other decompositions are exact. Decomposition is not used with arbitrary structuring elements, unless it is a flat structuring element whose neighborhood is all 1's.

To view the sequence of structuring elements used in a decomposition, use the `STREL` `getsequence` method. The `getsequence` function returns an array of the structuring elements that form the decomposition. For example, here are the structuring elements created in the decomposition of a diamond shaped structuring element.

```
sel = strel('diamond',4)
sel =
Flat STREL object containing 41 neighbors.
Decomposition: 3 STREL objects containing a total of 13 neighbors
```

```
Neighborhood:
     0     0     0     0     1     0     0     0     0
     0     0     0     1     1     1     0     0     0
     0     0     1     1     1     1     1     0     0
     0     1     1     1     1     1     1     1     0
     1     1     1     1     1     1     1     1     1
     0     1     1     1     1     1     1     1     0
     0     0     1     1     1     1     1     0     0
     0     0     0     1     1     1     0     0     0
     0     0     0     0     1     0     0     0     0

seq = getsequence(sel)
seq =
3x1 array of STREL objects

seq(1)
ans =
Flat STREL object containing 5 neighbors.

Neighborhood:
     0     1     0
     1     1     1
     0     1     0

seq(2)
ans =
Flat STREL object containing 4 neighbors.

Neighborhood:
     0     1     0
     1     0     1
     0     1     0

seq(3)
ans =
Flat STREL object containing 4 neighbors.
```

```
Neighborhood:
     0     0     1     0     0
     0     0     0     0     0
     1     0     0     0     1
     0     0     0     0     0
     0     0     1     0     0
```

Dilating an Image

To dilate an image, use the `imdilate` function. The `imdilate` function accepts two primary arguments:

- The input image to be processed (grayscale, binary, or packed binary image)
- A structuring element object, returned by the `strel` function, or a binary matrix defining the neighborhood of a structuring element

`imdilate` also accepts two optional arguments: `PADOPT` and `PACKOPT`. The `PADOPT` argument affects the size of the output image. The `PACKOPT` argument identifies the input image as packed binary. (See the `bwpack` reference page for information about binary image packing.)

This example dilates a simple binary image containing one rectangular object.

```
BW = zeros(9,10);
BW(4:6,4:7) = 1
BW =
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     1     1     1     1     0     0     0
     0     0     0     1     1     1     1     0     0     0
     0     0     0     1     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

To expand all sides of the foreground component, the example uses a 3-by-3 square structuring element object. (For more information about using the `strel` function, see “Structuring Elements” on page 9-7.)

```
SE = strel('square',3)
SE =

Flat STREL object containing 3 neighbors.

Neighborhood:
     1     1     1
     1     1     1
     1     1     1
```

To dilate the image, pass the image, `BW`, and the structuring element, `SE`, to the `imdilate` function. Note how dilation adds a rank of 1's to all sides of the foreground object.

```
BW2 = imdilate(BW,SE)
BW2 =
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     1     1     1     1     1     1     0     0
     0     0     1     1     1     1     1     1     0     0
     0     0     1     1     1     1     1     1     0     0
     0     0     1     1     1     1     1     1     0     0
     0     0     1     1     1     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

Eroding an Image

To erode an image, use the `imerode` function. The `imerode` function accepts two primary arguments:

- The input image to be processed (grayscale, binary, or packed binary image)
- A structuring element object, returned by the `strel` function, or a binary matrix defining the neighborhood of a structuring element

`imerode` also accepts three optional arguments: `PADOPT`, `PACKOPT`, and `M`.

The `PADOPT` argument affects the size of the output image. The `PACKOPT` argument identifies the input image as packed binary. If the image is packed binary, `M` identifies the number of rows in the original image. (See the `bwpack` reference page for more information about binary image packing.)

The following example erodes the binary image, `circbw.tif`:

1. Read the image into the MATLAB workspace.

```
BW1 = imread('circbw.tif');
```

2. Create a structuring element. The following code creates a diagonal structuring element object. (For more information about using the `strel` function, see "Structuring Elements" on page 9-7.)

```
SE = strel('arbitrary ,eye(5));
SE=

Flat STREL object containing 5 neighbors.

Neighborhood:
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

3. Call the `imerode` function, passing the image, `BW`, and the structuring element, `SE`, as arguments.

```
BW2 = imerode(BW1,SE);
```

Notice the diagonal streaks on the right side of the output image. These are due to the shape of the structuring element.

```
imshow(BW1)
figure, imshow(BW2)
```

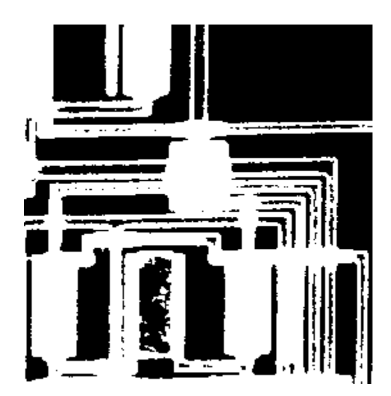



Figure 9-4: Circbw.tif Before and After Erosion with a Diagonal Structuring Element

Combining Dilation and Erosion

Dilation and erosion are often used in combination to implement image processing operations. For example, the definition of a morphological *opening* of an image is an erosion followed by a dilation, using the same structuring element for both operations. The related operation, morphological *closing* of an image is the reverse: it consists of dilation followed by an erosion with the same structuring element.

The following section uses `imdilate` and `imerode` to illustrate how to implement a morphological opening. Note, however, that the toolbox already includes the `imopen` function which performs this processing. The toolbox includes functions that perform many common morphological operations. See "Dilation- and Erosion-Based Functions" on page 9-16 for a complete list. To see some of the morphology functions used in an extended example, see "Example: Marker-Controlled Watershed Segmentation" on page 9-41.

Morphological Opening

You can use morphological opening to remove small objects from an image while preserving the shape and size of larger objects in the image. For example, you can use the `imopen` function to remove all the circuit lines from the original circuit image, `circbw.tif`, creating an output image that contains only the rectangular shapes of the microchips.

To morphologically open the image, perform these steps:

1 Create a structuring element.

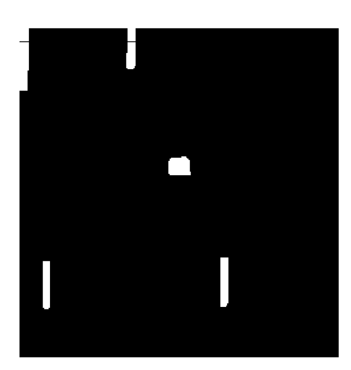
```
SE = strel('rectangle',[40 30]);
```

The structuring element should be large enough to remove the lines when you erode the image, but not large enough to remove the rectangles. It should consist of all 1's, so it removes everything but large continuous patches of foreground pixels.

2 Erode the image with the structuring element.

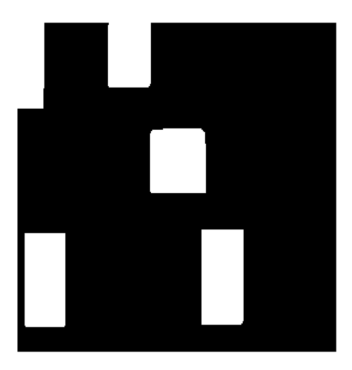
```
BW1 = imread('circbw.tif');
BW2 = imerode(BW1,SE);
imshow(BW2)
```

This removes all of the lines, but also shrinks the rectangles.



3 To restore the rectangles to their original size, dilate the eroded image using the same structuring element, `SE`.

```
BW3 = imdilate(BW2,SE);
imshow(BW3)
```



Dilation- and Erosion-Based Functions

This section describes two common image processing operations that are based on dilation and erosion:

- Skeletonization
- Perimeter Determination

This table lists other functions in the toolbox that perform common morphological operations that are based on dilation and erosion. For more information about these functions, see their reference pages.

Table 9-3: Dilation- and Erosion-Based Functions

Function	Morphological Definition
`bwhitmiss`	Logical `AND` of an image, eroded with one structuring element, and the image's complement, eroded with a second structuring element.
`imbothat`	Subtracts the original image from a morphologically closed version of the image. Can be used to find intensity troughs in an image.
`imclose`	Dilates an image, and then erodes the dilated image using the same structuring element for both operations.

Table 9-3: Dilation- and Erosion-Based Functions

Function	Morphological Definition
`imopen`	Erodes an image and then dilates the eroded image using the same structuring element for both operations.
`imtophat`	Subtracts a morphologically opened image from the original image. Can be used to enhance contrast in an image.

Skeletonization

To reduce all objects in an image to lines, without changing the essential structure of the image, use the `bwmorph` function. This process is known as *skeletonization*.

```
BW1 = imread('circbw.tif');
BW2 = bwmorph(BW1,'skel',Inf);
imshow(BW1)
figure, imshow(BW2)
```



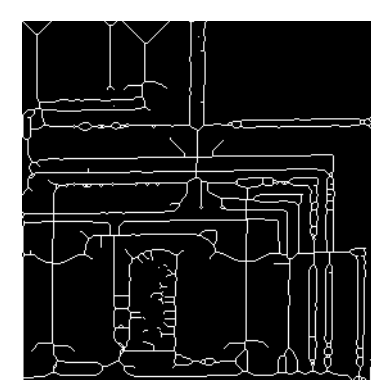


Figure 9-5: Circbw.tif Before and After Skeletonization

Perimeter Determination

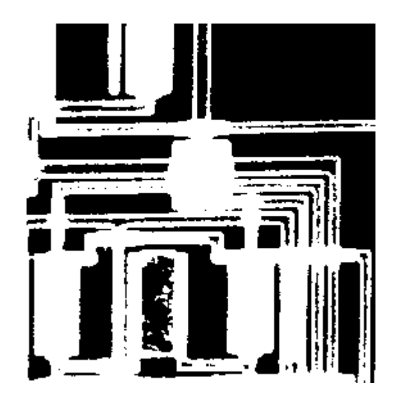
The `bwperim` function determines the perimeter pixels of the objects in a binary image. A pixel is considered a perimeter pixel if it satisfies both of these criteria:

- The pixel is `on`.

- One (or more) of the pixels in its neighborhood is `off`.

For example, this code finds the perimeter pixels in a binary image of a circuit board.

```
BW1 = imread('circbw.tif');
BW2 = bwperim(BW1);
imshow(BW1)
figure, imshow(BW2)
```



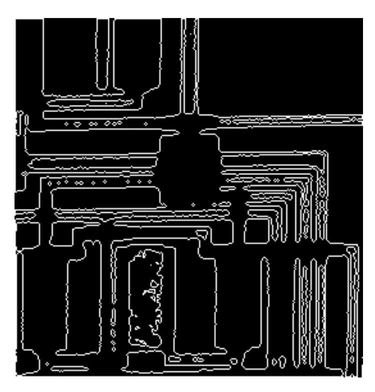


Figure 9-6: Circbw.tif Before and After Perimeter Determination

Morphological Reconstruction

Morphological reconstruction is another major part of morphological image processing. Based on dilation, morphological reconstruction has these unique properties:

- Processing is based on two images, a marker and a mask, rather than one image and a structuring element
- Processing repeats until stability; i.e., the image no longer changes
- Processing is based on the concept of connectivity, rather than a structuring element.

The following sections describe the functions in the toolbox that are based on morphological reconstruction:

- "Marker and Mask" – This section describes the fundamental reconstruction function, `imreconstruct`, and provides background information about morphological reconstruction.
- "Pixel Connectivity" – This section describes how pixel connectivity affects morphological reconstruction.
- "Flood-Fill Operations" – This section describes how to use the `imfill` function, which is based on morphological reconstruction.
- "Finding Peaks and Valleys" – This section describes a group of functions, all based on morphological reconstruction, that process image extrema, i.e., the areas of high- and low-intensity in images.

Marker and Mask

Morphological reconstruction processes one image, called the *marker*, based on the characteristics of another image, called the *mask*. The high-points, or peaks, in the marker image specify where processing begins. The processing continues until the image values stop changing.

To illustrate morphological reconstruction, consider this simple image. It contains two primary regions, the blocks of pixels containing the value `14` and `18`. The background is primarily all set to `10`, with some pixels set to `11`.

```
A = [10    10    10    10    10    10    10    10    10    10;
     10    14    14    14    10    10    11    10    11    10;
     10    14    14    14    10    10    10    11    10    10;
     10    14    14    14    10    10    11    10    11    10;
     10    10    10    10    10    10    10    10    10    10;
     10    11    10    10    10    18    18    18    10    10;
     10    10    10    11    10    18    18    18    10    10;
     10    10    11    10    10    18    18    18    10    10;
     10    11    10    11    10    10    10    10    10    10;
     10    10    10    10    10    10    11    10    10    10];
```

To morphologically reconstruct this image, perform these steps:

1 Create a marker image. As with the structuring element in dilation and erosion, the characteristics of the marker image determine the processing performed in morphological reconstruction. The peaks in the marker image should identify the location of objects in the mask image that you want to emphasize.

One way to create a marker image is to subtract a constant from the mask image, using `imsubtract`.

```
marker = imsubtract(A,2)
marker =
     8     8     8     8     8     8     8     8     8     8
     8    12    12    12     8     8     9     8     9     8
     8    12    12    12     8     8     8     9     8     8
     8    12    12    12     8     8     9     8     9     8
     8     8     8     8     8     8     8     8     8     8
     8     9     8     8     8    16    16    16     8     8
     8     8     8     9     8    16    16    16     8     8
     8     8     9     8     8    16    16    16     8     8
     8     9     8     9     8     8     8     8     8     8
     8     8     8     8     8     8     9     8     8     8
```

2 Call the `imreconstruct` function to morphologically reconstruct the image. In the output image, note how all the intensity fluctuations except the intensity peak have been removed.

```
recon = imreconstruct(marker, mask)
recon =
    10    10    10    10    10    10    10    10    10    10
    10    12    12    12    10    10    10    10    10    10
    10    12    12    12    10    10    10    10    10    10
    10    12    12    12    10    10    10    10    10    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    16    16    16    10    10
    10    10    10    10    10    16    16    16    10    10
    10    10    10    10    10    16    16    16    10    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    10    10    10    10    10
```

Understanding Morphological Reconstruction

Morphological reconstruction can be thought of conceptually as repeated dilations of the marker image until the contour of the marker image fits under the mask image. In this way, the peaks in the marker image "spread out", or dilate.

This figure illustrates this processing in 1-D. Each successive dilation is constrained to lie underneath the mask. When further dilation ceases to change the image, processing stops. The final dilation is the reconstructed image. (Note: the actual implementation of this operation in the toolbox is done much more efficiently. See the `imreconstruct` reference page for more details.) The figure shows the successive dilations of the marker.

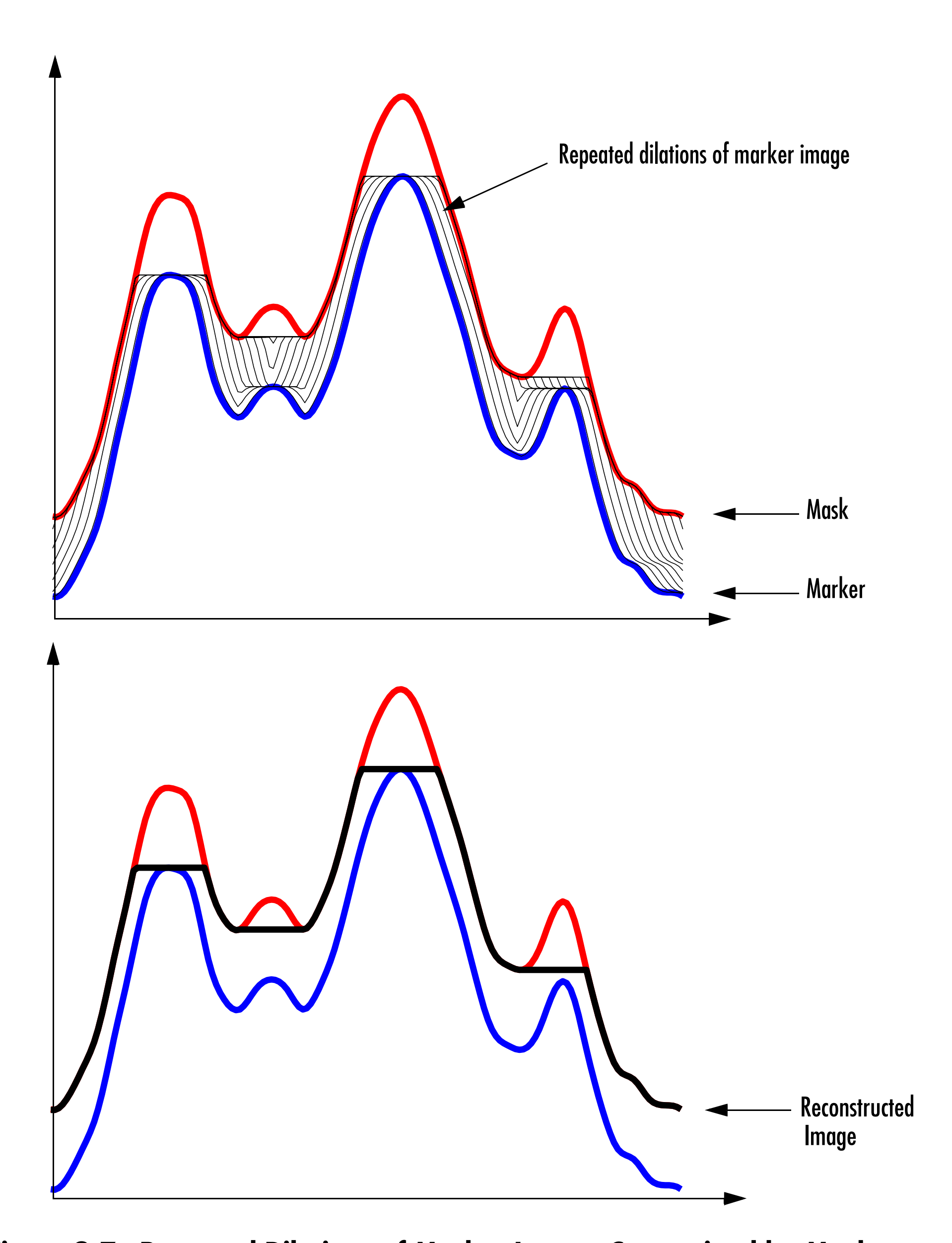


Figure 9-7: Repeated Dilations of Marker Image, Constrained by Mask

Pixel Connectivity

Morphological processing starts at the peaks in the marker image and spreads throughout the rest of the image based on the connectivity of the pixels. Connectivity defines which pixels are connected to other pixels.

For example, this binary image contains one foreground object—all the pixels that are set to 1. If the foreground is 4-connected, the image has one background object, all the pixels are set to 0. However, if the foreground is 8-connected, the foreground makes a closed loop and the image has two separate background objects: the pixels in the loop and the pixels outside the loop.

```
0    0    0    0    0    0    0    0
0    1    1    1    1    1    0    0
0    1    0    0    0    1    0    0
0    1    0    0    0    1    0    0
0    1    0    0    0    1    0    0
0    1    1    1    1    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
```

Defining Connectivity in an Image

The following table lists all the standard two- and three-dimensional connectivities supported by the toolbox. See these sections for more information:

- “Choosing a Connectivity”
- “Specifying Custom Connectivities”

Table 9-4: Supported Connectivities

Two-Dimensional Connectivities		
4-connected	Pixels are connected if their edges touch. This means that a pair of adjoining pixels are part of the same object only if they are both on and are connected along the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. This means that if two adjoining pixels are on, they are part of the same object, regardless of whether they are connected along the horizontal, vertical, or diagonal direction.	
Three-Dimensional Connectivities		
6-connected	Pixels are connected if their faces touch.	6 faces
18-connected	Pixels are connected if their faces or edges touch.	6 faces + 12 edges
26-connected	Pixels are connected if their faces, edges, or corners touch.	6 faces + 12 edges + 8 corners

Choosing a Connectivity

The type of neighborhood you choose affects the number of objects found in an image and the boundaries of those objects. For this reason, the results of many morphology operations often differ depending upon the type connectivity you specify.

For example, if you specify a 4-connected neighborhood, this binary image contains two objects; if you specify an 8-connected neighborhood, the image has one object.

```
0     0     0     0     0     0
0     1     1     0     0     0
0     1     1     0     0     0
0     0     0     1     1     0
0     0     0     1     1     0
```

Specifying Custom Connectivities

You can also define custom neighborhoods by specifying a 3-by-3-by-...-by-3 array of 0's and 1's. The 1-valued elements define the connectivity of the neighborhood relative to the center element.

For example, this array defines a "North/South" connectivity that has the effect of breaking up an image into independent columns.

```
CONN = [ 0 1 0; 0 1 0; 0 1 0 ]
CONN =
     0     1     0
     0     1     0
     0     1     0
```

Note Connectivity arrays must be symmetric about their center element. Also, you can use 2-D connectivity array with a 3-D image; the connectivity affects each "page" in the 3-D image.

Flood-Fill Operations

The `imfill` function performs a *flood-fill* operation on binary and grayscale images. For binary images, `imfill` changes connected background pixels (0s) to foreground pixels (1s), stopping when it reaches object boundaries. For grayscale images, `imfill` brings the intensity values of dark areas that are surrounded by lighter areas up to the same intensity level as surrounding pixels. (In effect, `imfill` removes regional minima that are not connected to the image border. See "Finding Areas of High- or Low-Intensity" for more information.) This operation can be useful in removing irrelevant artifacts from images.

This section includes information about:

- Specifying the connectivity in flood-fill operations
- Specifying the starting point for binary image fill operations
- Filling holes in binary or grayscale images

Specifying Connectivity

For both binary and grayscale images, the boundary of the fill operation is determined by the connectivity you specify.

Note `imfill` differs from the other object-based operations in that it operates on *background* pixels. When you specify connectivity with `imfill`, you are specifying the connectivity of the background, not the foreground.

The implications of connectivity can be illustrated with this matrix.

```
BW = [ 0    0    0    0    0    0    0    0;
       0    1    1    1    1    1    0    0;
       0    1    0    0    0    1    0    0;
       0    1    0    0    0    1    0    0;
       0    1    0    0    0    1    0    0;
       0    1    1    1    1    0    0    0;
       0    0    0    0    0    0    0    0;
       0    0    0    0    0    0    0    0];
```

If the background is 4-connected, this binary image contains two separate background elements (the part inside the loop and the part outside). If the

background is 8-connected, the pixels connect diagonally, and there is only one background element.

Specifying the Starting Point

For binary images, you can specify the starting point of the fill operation by passing in the location subscript or by using `imfill` in interactive mode, selecting starting pixels with a mouse. See the reference page for `imfill` for more information using `imfill` interactively.

For example, if you call `imfill`, specifying the pixel `BW(4,3)` as the starting point, `imfill` only fills in the inside of the loop because, by default, the background is 4-connected.

```
imfill(BW,[4 3])

ans =
     0     0     0     0     0     0     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

If you specify the same starting point, but use an 8-connected background connectivity, `imfill` fills in the entire image.

```
imfill(BW,[4 3],8)

ans =
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
```

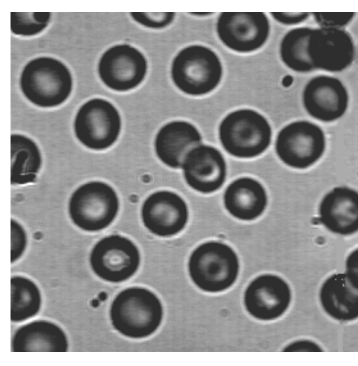
Filling Holes

A common use of the flood-fill operation is to fill “holes” in images. For example, suppose you have an image, binary or grayscale, in which the foreground objects represent spheres. In the image, these objects should appear as disks, but instead are donut shaped because of reflections in the original photograph. Before doing any further processing of the image, you may want to first fill in the “donut holes” using `imfill`.

Because the use of flood-fill to fill holes is so common, `imfill` includes special syntax to support it for both binary and grayscale images. In this syntax, you just specify the argument `'holes'`; you do not have to specify starting locations in each hole.

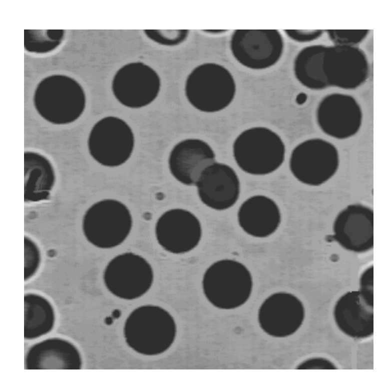
To illustrate, this example fills holes in a grayscale image of blood cells.

```
blood = imread('blood1.tif');
imshow(blood)
blood2 = imcomplement(imfill(imcomplement(blood),'holes'));
figure, imshow(blood2)
```





Original



After Filling Holes

Finding Peaks and Valleys

Grayscale images can be thought of in three-dimensions: the *x*- and *y*-axes represent pixel positions and the *z*-axes represents the intensity of each pixel. In this interpretation, the intensity values represent elevations as in a topographical map. The areas of high-intensity and low-intensity in an image, peaks and valleys in topographical terms, can be important morphological features because they often mark relevant image objects.

For example, in an image of several spherical objects, points of high intensity could represent the tops of the objects. Using morphological processing, these maxima can be used to identify objects in an image.

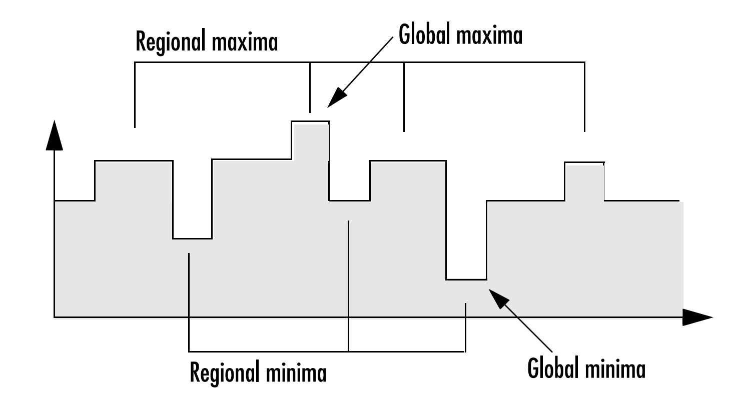
This section covers these topics:

- "Understanding the Maxima and Minima Functions"
- "Finding Areas of High- or Low-Intensity"
- "Suppressing Minima and Maxima"
- "Imposing a Minima"

Understanding the Maxima and Minima Functions

An image can have multiple regional maxima or minima but only a single global maxima or minima. Determining image peaks or valleys can be used to create marker images that are used in morphological reconstruction. For a detailed example, see "Example: Marker-Controlled Watershed Segmentation" on page 9-41.

This figure illustrates the concept in 1-D.



Finding Areas of High- or Low-Intensity

The toolbox includes functions that you can use to find areas of high- or low-intensity in an image:

- The `imregionalmax` and `imregionalmin` functions identify *all* regional minima or maxima.
- The `imextendedmax` and `imextendedmin` functions identify all regional minima or maxima that are greater than or less than a specified threshold.

The functions accept a grayscale image as input and return a binary image as output. In the output binary image, the regional minima or maxima are set to `1`; all other pixels are set to `0`.

For example, this simple image contains two primary regional maxima, the blocks of pixels containing the value 13 and 18, and several smaller maxima, set to 11.

```
A = [10   10   10   10   10   10   10   10   10   10;
     10   13   13   13   10   10   11   10   11   10;
     10   13   13   13   10   10   10   11   10   10;
     10   13   13   13   10   10   11   10   11   10;
     10   10   10   10   10   10   10   10   10   10;
     10   11   10   10   10   18   18   18   10   10;
     10   10   10   11   10   18   18   18   10   10;
     10   10   11   10   10   18   18   18   10   10;
     10   11   10   11   10   10   10   10   10   10;
     10   10   10   10   10   10   11   10   10   10]
```

The binary image returned by imregionalmax pinpoints all these regional maxima.

```
B = imregionalmax(A)
B =
     0    0    0    0    0    0    0    0    0    0
     0    1    1    1    0    0    1    0    1    0
     0    1    1    1    0    0    0    1    0    0
     0    1    1    1    0    0    1    0    1    0
     0    0    0    0    0    0    0    0    0    0
     0    1    0    0    0    1    1    1    0    0
     0    0    0    1    0    1    1    1    0    0
     0    0    1    0    0    1    1    1    0    0
     0    1    0    1    0    0    0    0    0    0
     0    0    0    0    0    0    1    0    0    0
```

You may want to only identify areas of the image where the change in intensity is extreme; that is, the difference between the pixel and neighboring pixels is greater than (or less than) a certain threshold. For example, to find only those regional maxima in the sample image, A, that are at least two units higher than their neighbors, use imextendedmax.

```
B = imextendedmax(A,2)
B =
     0     0     0     0     0     0     0     0     0     0
     0     1     1     1     0     0     0     0     0     0
     0     1     1     1     0     0     0     0     0     0
     0     1     1     1     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

Suppressing Minima and Maxima

In an image, every small fluctuation in intensity represents a regional minima or maxima. You may only be interested in significant minima or maxima and not in these smaller minima and maxima caused by background texture.

To remove the less significant minima and maxima but retain the significant minima and maxima, use the `imhmax` or `imhmin` functions. With these functions, you can specify a contrast criteria or threshold level, h, that suppresses all maxima whose height is less than h or whose minima are greater than h.

Note The `imregionalmin`, `imregionalmax`, `imextendedmin` and `imextendedmax` functions return a binary image that marks the locations of the regional minima and maxima in an image. The `imhmax` and `imhmin` functions produce an altered image.

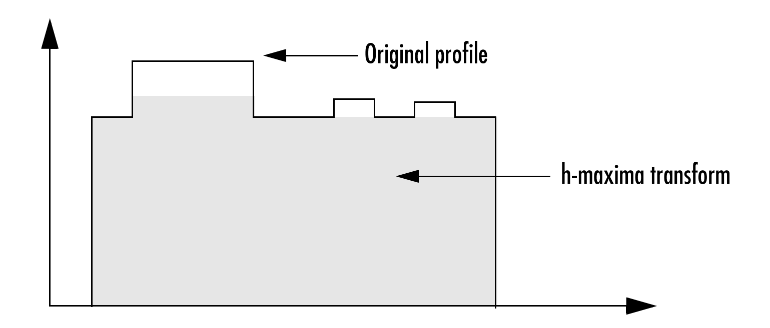
For example, this simple image contains two primary regional maxima, the blocks of pixels containing the value `14` and `18`, and several smaller maxima, set to `11`.

```
A = [10   10   10   10   10   10   10   10   10   10;
     10   14   14   14   10   10   11   10   11   10;
     10   14   14   14   10   10   10   11   10   10;
     10   14   14   14   10   10   11   10   11   10;
     10   10   10   10   10   10   10   10   10   10;
     10   11   10   10   10   18   18   18   10   10;
     10   10   10   11   10   18   18   18   10   10;
     10   10   11   10   10   18   18   18   10   10;
     10   11   10   11   10   10   10   10   10   10;
     10   10   10   10   10   10   11   10   10   10];
```

To eliminate all regional maxima except the two significant maxima, use `imhmax`, specifying a threshold value of `2`. Note that `imhmax` only affects the maxima; none of the other pixel values are changed. The two significant maxima remain, although their heights are reduced.

```
B = imhmax(A,2)
B =
     10   10   10   10   10   10   10   10   10   10
     10   12   12   12   10   10   10   10   10   10
     10   12   12   12   10   10   10   10   10   10
     10   12   12   12   10   10   10   10   10   10
     10   10   10   10   10   10   10   10   10   10
     10   10   10   10   10   16   16   16   10   10
     10   10   10   10   10   16   16   16   10   10
     10   10   10   10   10   16   16   16   10   10
     10   10   10   10   10   10   10   10   10   10
     10   10   10   10   10   10   10   10   10   10
```

This figure takes the second row from the sample image to illustrate in 1-D how `imhmax` changes the profile of the image.



Imposing a Minima

You can emphasize specific minima (dark objects) in an image using the `imimposemin` function. The `imimposemin` function uses morphological reconstruction to eliminate all minima from an the image except the minima you specify.

To illustrate the process of imposing a minima, this code creates a simple image containing two primary regional minima and several other regional minima.

```
mask = uint8(10*ones(10,10));
mask(6:8,6:8) = 2;
mask(2:4,2:4) = 7;
mask(3,3) = 5;
mask(2,9) = 9
mask(3,8) = 9
mask(9,2) = 9
mask(8,3) = 9
```

```
mask = 10    10    10    10    10    10    10    10    10    10
       10     7     7     7    10    10    10    10     9    10
       10     7     6     7    10    10    10     9    10    10
       10     7     7     7    10    10    10    10    10    10
       10    10    10    10    10    10    10    10    10    10
       10    10    10    10    10     2     2     2    10    10
       10    10    10    10    10     2     2     2    10    10
       10    10     9    10    10     2     2     2    10    10
       10     9    10    10    10    10    10    10    10    10
       10    10    10    10    10    10    10    10    10    10
```

Creating a Marker Image

To obtain an image that emphasizes the two deepest minima and removes all others, create a marker image that pinpoints the two minima of interest. You can create the marker image by explicitly setting certain pixels to specific values or by using other morphological functions to extract the features you want to emphasize in the mask image.

This example uses `imextendedmin` to get a binary image that shows the locations of the two deepest minima.

```
marker = imextendedmin(mask,1)
marker = 0    0    0    0    0    0    0    0    0    0
         0    0    0    0    0    0    0    0    0    0
         0    0    1    0    0    0    0    0    0    0
         0    0    0    0    0    0    0    0    0    0
         0    0    0    0    0    0    0    0    0    0
         0    0    0    0    0    1    1    1    0    0
         0    0    0    0    0    1    1    1    0    0
         0    0    0    0    0    1    1    1    0    0
         0    0    0    0    0    0    0    0    0    0
         0    0    0    0    0    0    0    0    0    0
```

Applying the Marker Image to the Mask

Now use `imimposemin` to create new minima in the mask image at the points specified by the marker image. Note how `imimposemin` sets the values of pixels specified by the marker image to the lowest value supported by the datatype (`0` for `uint8` values). `imimposemin` also changes the values of all the other pixels in the image to eliminate the other minima.

```
I = imimposemin(mask,marker)
I =
   11   11   11   11   11   11   11   11   11   11
   11    8    8    8   11   11   11   11   11   11
   11    8    0    8   11   11   11   11   11   11
   11    8    8    8   11   11   11   11   11   11
   11   11   11   11   11   11   11   11   11   11
   11   11   11   11   11    0    0    0   11   11
   11   11   11   11   11    0    0    0   11   11
   11   11   11   11   11    0    0    0   11   11
   11   11   11   11   11   11   11   11   11   11
   11   11   11   11   11   11   11   11   11   11
```

This figure illustrates in 1-D how `imimposemin` changes the profile of row 2 of the image.

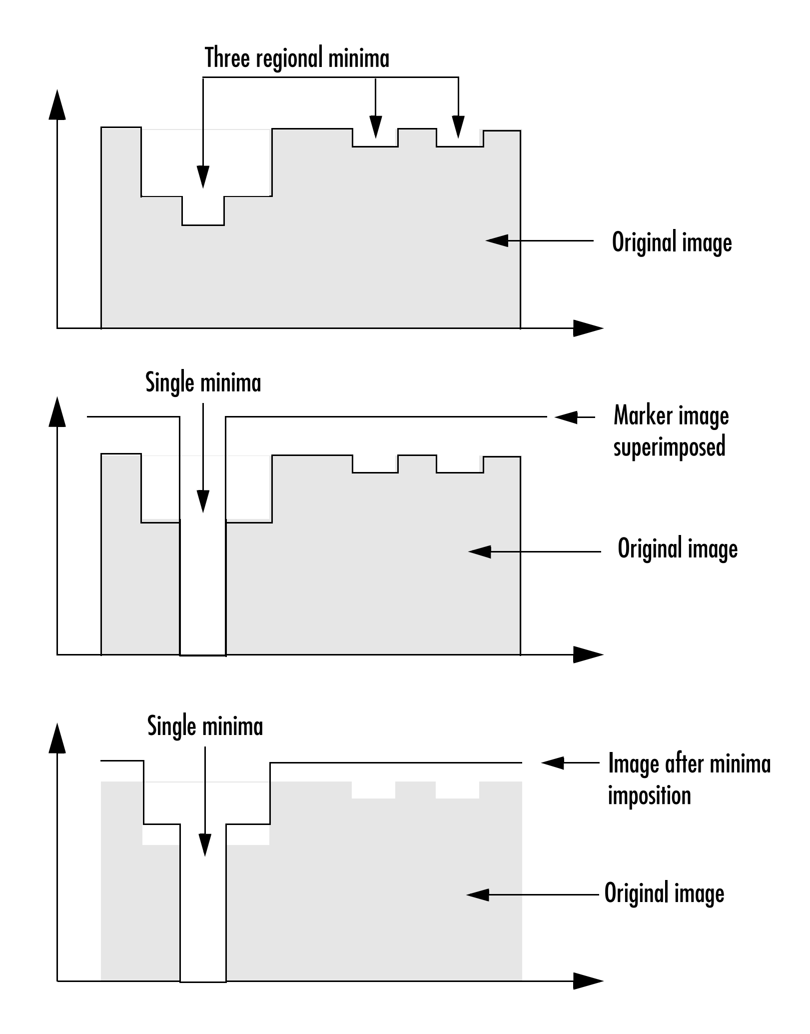


Figure 9-8: Imposing a Minima

Distance Transform

The distance transform provides a metric or measure of the separation of points in the image. The Image Processing Toolbox provides a function, `bwdist`, that calculates the distance between each pixel that is set to `off` (`0`) and the nearest nonzero pixel for binary images.

The `bwdist` function supports several distance metrics, listed in Table 9-5, Distance Metrics.

Table 9-5: Distance Metrics

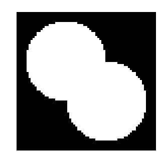
Distance Metric	Description	Illustration
Euclidean	The Euclidean distance is the straight-line distance between two pixels.	Image: 0 0 0 / 0 1 0 / 0 0 0; Distance transform: 1.41 1.0 1.41 / 1.0 0.0 1.0 / 1.41 1.0 1.41
City Block	The City Block distance metric measures the path between the pixels based on a 4-connected neighborhood. Pixels whose edges touch are 1 unit apart; pixels diagonally touching are 2 units apart.	Image: 0 0 0 / 0 1 0 / 0 0 0; Distance transform: 2 1 2 / 1 0 1 / 2 1 2

Table 9-5: Distance Metrics (Continued)

Distance Metric	Description	Illustration
Chessboard	The Chessboard distance metric measures the path between the pixels based on an 8-connected neighborhood. Pixels whose edges or corners touch are 1 unit apart.	0 0 0 0 1 0 0 0 0 Image 1 1 1 1 1 1 1 1 1 Distance transform
Quasi-Euclidean	The Quasi-Euclidean metric measures the total Euclidean distance along a set of horizontal, vertical, and diagonal line segments.	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 Image 2.8 2.2 2.0 2.2 2.8 2.2 1.4 1.0 1.4 2.2 2.0 1.0 0 1.0 2.0 2.2 1.4 1.0 1.4 2.2 2.8 2.2 2.0 2.2 2.8 Distance transform

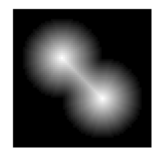
This example creates a binary image containing two overlapping circular objects.

```
center1 = -10;
center2 = -center1;
dist = sqrt(2*(2*center1)^2);
radius = dist/2 * 1.4;
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];
[x,y] = meshgrid(lims(1):lims(2));
bw1 = sqrt((x-center1).^2 + (y-center1).^2) <= radius;
bw2 = sqrt((x-center2).^2 + (y-center2).^2) <= radius;
bw = bw1 | bw2;
figure, imshow(bw), title('bw')
```



To compute the distance transform of the complement of the binary image, use the bwdist function. In the image of the distance transform, note how the center of the two circular areas are white.

```
D = bwdist(~bw);
figure, imshow(D,[]), title('Distance transform of ~bw')
```



Example: Marker-Controlled Watershed Segmentation

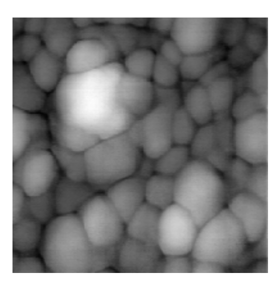
This example illustrates how to use many different morphology functions in combination to accomplish an image processing task: segmentation. In segmentation, objects in an image that are touching each other are divided into separate objects. (To see other extended examples, view the Image Processing Toolbox Morphology demos.)

The example performs these steps:

Step 1: Read in Images

Read in the `'afmsurf.tif'` image, which is an atomic force microscope image of a surface coating.

```
afm = imread('afmsurf.tif');
figure, imshow(afm), title('Surface Image');
```



The image contains many objects of different sizes that are touching each other. Object detection in an image is an example of image segmentation. To

segment touching objects, the Watershed transform is often used. If you view an image as a surface, with mountains (high intensity) and valleys (low intensity), the Watershed transform finds intensity valleys in an image.

To get the best result, maximize the contrast of the objects of interest to minimize the number of valleys found by the Watershed transform. A common technique for contrast enhancement is the combined use of the top-hat and bottom-hat transforms.

The top-hat transform is defined as the difference between the original image and its opening. The opening of an image is the collection of foreground parts of an image that fit a particular structuring element. The bottom-hat transform is defined as the difference between the closing of the original image and the original image. The closing of an image is the collection of background parts of an image that fit a particular structuring element.

Step 2: Create the Structuring Element

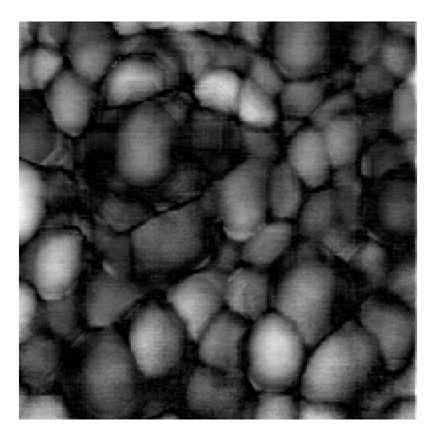
Since the objects of interest in our image look like disks, the example uses the `strel` function to create a disk structuring element. The size of the disk is based on an estimation of the average radius of the objects in the image.

```
se = strel('disk', 15);
```

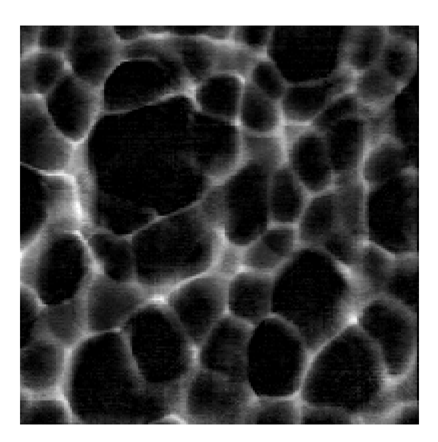
Step 3: Enhance the Image Contrast

The `imtophat` and `imbothat` functions return the top-hat and bottom-hat transforms, respectively, of the original image.

```
Itop = imtophat(afm, se);
Ibot = imbothat(afm, se);
figure, imshow(Itop, []), title('top-hat image');
```



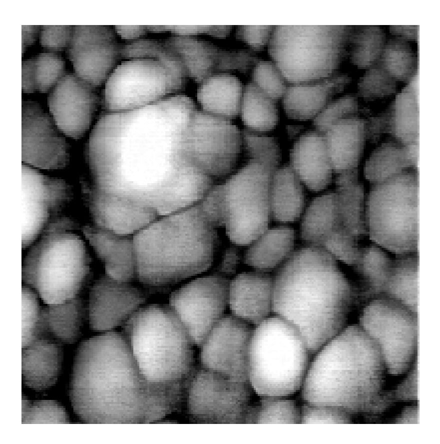
```
figure, imshow(Ibot, []), title('bottom-hat image');
```



Step 4: Exaggerate the Gaps Between Objects

The top-hat image contains the “peaks” of objects that fit the structuring element. The `imbothat` function shows the gaps between the objects. To maximize the contrast between the objects and the gaps that separate them from each other, the example adds the top-hat image to the original image, and then subtracts the “bottom-hat” image from the result. The example uses the toolbox image arithmetic functions, `imadd` and `imsubtract`, to perform these operations.

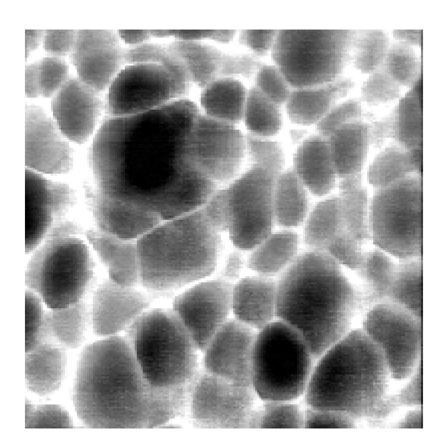
```
Ienhance = imsubtract(imadd(Itop, afm), Ibot);
figure, imshow(Ienhance), title('original + top-hat - bottom-hat');
```



Step 5: Convert Objects of Interest

Because the watershed transform detects intensity “valleys” in an image, the example uses the `imcomplement` function on the enhanced image to highlight the intensity valleys.

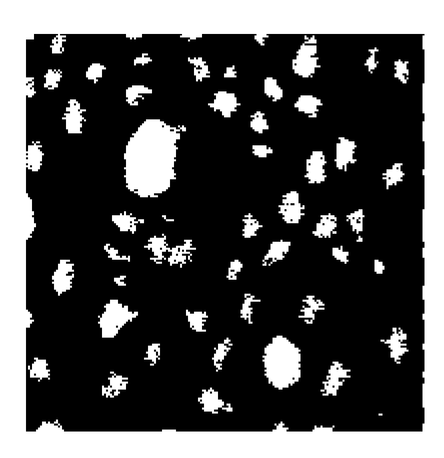
```
Iec = imcomplement(Ienhance);
figure, imshow(Iec), title('complement of enhanced image');
```



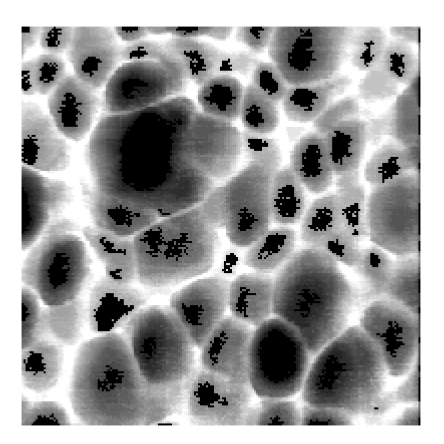
Step 6: Detect Intensity Valleys

The example detects all the intensity valleys deeper than a particular threshold with the `imextendedmin` function. The output of the `imextendedmin` function is a binary image. The location rather than the size of the regions in the `imextendedmin` image is important. The `imimposemin` function modifies the image to contain only those valleys found by the `imextendedmin` function. The `imimposemin` function also changes a valley's pixel values to zero (deepest possible valley for `uint8` images). All regions containing an imposed minima are detected by the watershed transform.

```
Iemin = imextendedmin(Iec, 22);
Iimpose = imimposemin(Iec, Iemin);
figure, imshow(Iemin), title('extended minima image');
```



```
figure, imshow(Iimpose), title('imposed minima image');
```



Step 7: Watershed Segmentation

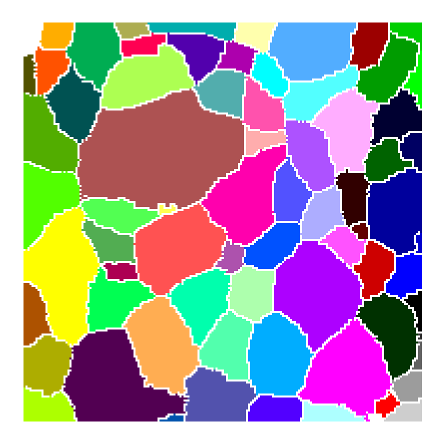
Use the `watershed` function to accomplish Watershed segmentation of the imposed minima image.

```
wat = watershed(Iimpose);
```

The `watershed` function returns a label matrix containing nonnegative numbers that correspond to watershed regions. Pixels that do not fall into any watershed region are given a pixel value of `0`.

A good way to visualize a label matrix is to convert it to a color image, using the `label2rgb` function. In the color version of the image, each labeled region displays in a different color and the pixels that separate the regions display white.

```
rgb = label2rgb(wat);
figure, imshow(rgb);
title('watershed segmented image');
```

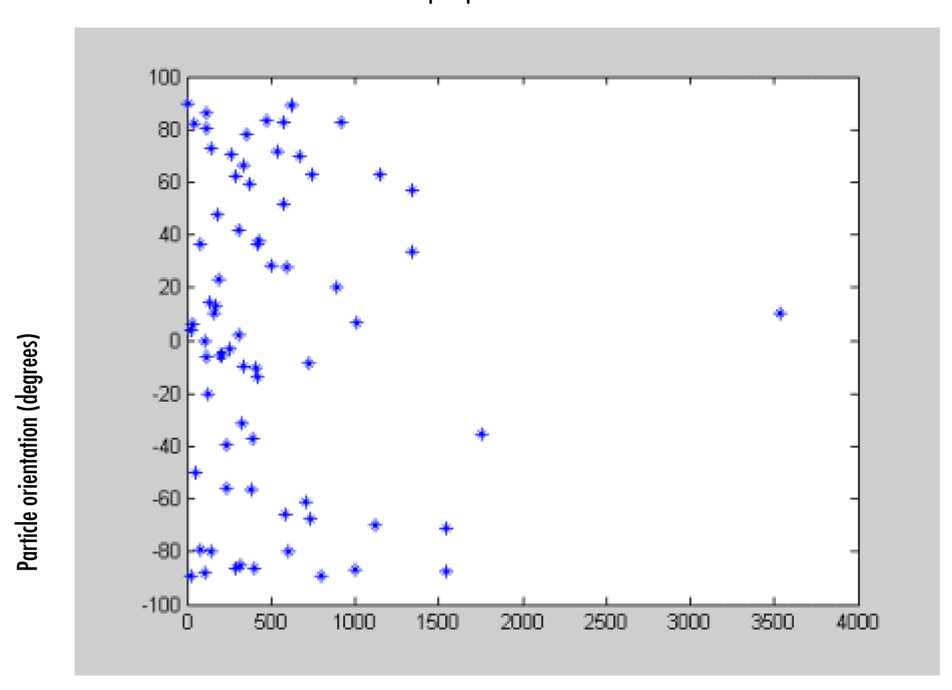


Step 8: Extract Features from Label Matrix

Features can be extracted from the label matrix with the `regionprops` function. For example, we can calculate two measurements (area and orientation) and view them as a function of one another.

```
stats = regionprops(wat, 'Area', 'Orientation');
area = [stats.Area];
orient = [stats.Orientation];
figure, plot(area, orient, 'b*');
title('Relationship of Particle Orientation to Area');
xlabel('particle area (pixels)');
ylabel('particle orientation (degrees)');
```

Relationship of particle orientation to area



Particle area (pixels)

Objects, Regions, and Feature Measurement

The toolbox includes several functions that return information about the features in a binary image, including:

- Connected-component labeling, and using the label matrix to get statistics about an image
- Selecting objects in an image
- Finding the area of a binary image
- Finding the Euler number of a binary image

Connected-Component Labeling

The `bwlabel` and the `bwlabeln` functions perform *connected-component labeling*, which is a method for identifying each object in a binary image. The `bwlabel` function supports 2-D inputs only; the `bwlabeln` function supports inputs of any dimension.

These functions return a matrix, called a *label matrix*. A label matrix is an image, the same size as the input image, in which the objects in the input image are distinguished by different integer values in the output matrix. For example, `bwlabel` can identify the objects in this binary image.

```
BW = [0     0     0     0     0     0     0     0;
      0     1     1     0     0     1     1     1;
      0     1     1     0     0     0     1     1;
      0     1     1     0     0     0     0     0;
      0     0     0     1     1     0     0     0;
      0     0     0     1     1     0     0     0;
      0     0     0     1     1     0     0     0;
      0     0     0     0     0     0     0     0];
```

```
X = bwlabel(BW,4)
X =
     0     0     0     0     0     0     0     0
     0     1     1     0     0     3     3     3
     0     1     1     0     0     0     3     3
     0     1     1     0     0     0     0     0
     0     0     0     2     2     0     0     0
     0     0     0     2     2     0     0     0
     0     0     0     2     2     0     0     0
     0     0     0     0     0     0     0     0
```

In the output matrix, the 1's represent one object, the 2's a second object, and the 3's a third. (If you had used 8-connected neighborhoods (the default), there would be only two objects, because the first and second objects would be a single object, connected along the diagonal.)

Viewing a Label Matrix

The label matrix returned by `bwlabel` or `bwlabeln` is of class `double`; it is not a binary image. One way to view it is to display it as a pseudo-color indexed image, using `label2rgb`. In the pseudo-color image, the number that identifies each object in the label matrix maps to a different color in the associated colormap matrix. When you view a label matrix as an RGB image, the objects in the image are easier to distinguish.

To illustrate this technique, this example uses `label2rgb` to view the label matrix, `X`. The call to `label2rgb` specifies one of the standard MATLAB colormaps, `jet`. The third argument, `'k'`, specifies the background color (black).

```
X = bwlabel(BW1,4);
RGB = label2rgb(X, @jet, 'k');
imshow(RGB,'notruesize')
```

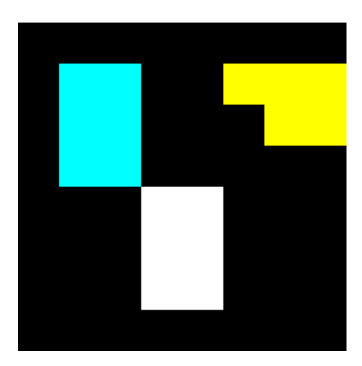


Figure 9-9: Using Color to Distinguish Objects in a Binary Image

Selecting Objects in a Binary Image

You can use the `bwselect` function to select individual objects in a binary image. You specify pixels in the input image, and `bwselect` returns a binary image that includes only those objects from the input image that contain one of the specified pixels.

You can specify the pixels either noninteractively or with a mouse. For example, suppose you want to select objects in the image displayed in the current axes. You type

```
BW2 = bwselect;
```

The cursor changes to a cross-hair when it is over the image. Click on the objects you want to select; `bwselect` displays a small star over each pixel you click on. When you are done, press **Return**. `bwselect` returns a binary image consisting of the objects you selected, and removes the stars.

See the reference page for `bwselect` for more information.

Finding the Area of Binary Images

The `bwarea` function returns the area of a binary image. The area is a measure of the size of the foreground of the image. Roughly speaking, the area is the number of `on` pixels in the image.

`bwarea` does not simply count the number of pixels set to `on`, however. Rather, `bwarea` weights different pixel patterns unequally when computing the area. This weighting compensates for the distortion that is inherent in representing

a continuous image with discrete pixels. For example, a diagonal line of 50 pixels is longer than a horizontal line of 50 pixels. As a result of the weighting bwarea uses, the horizontal line has area of 50, but the diagonal line has area of 62.5.

This example uses bwarea to determine the percentage area increase in circbw.tif that results from a dilation operation.

```
BW = imread('circbw.tif');
SE = ones(5);
BW2 = imdilate(BW,SE);
increase = (bwarea(BW2) - bwarea(BW))/bwarea(BW);
increase =

    0.3456
```

See the reference page for bwarea for more information about the weighting pattern.

Finding the Euler Number of a Binary Image

The bweuler function returns the Euler number for a binary image. The Euler number is a measure of the topology of an image. It is defined as the total number of objects in the image minus the number of holes in those objects. You can use either 4- or 8-connected neighborhoods.

This example computes the Euler number for the circuit image, using 8-connected neighborhoods.

```
BW1 = imread('circbw.tif');
eul = bweuler(BW1,8)

eul =

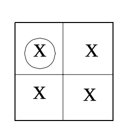
   -85
```

In this example, the Euler number is negative, indicating that the number of holes is greater than the number of objects.

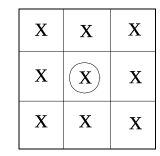
Lookup Table Operations

Certain binary image operations can be implemented most easily through lookup tables. A lookup table is a column vector in which each element represents the value to return for one possible combination of pixels in a neighborhood.

You can use the makelut function to create lookup tables for various operations. makelut creates lookup tables for 2-by-2 and 3-by-3 neighborhoods. This figure illustrates these types of neighborhoods. Each neighborhood pixel is indicated by an x, and the center pixel is the one with a circle.



2-by-2 neighborhood



3-by-3 neighborhood

For a 2-by-2 neighborhood, there are 16 possible permutations of the pixels in the neighborhood. Therefore, the lookup table for this operation is a 16-element vector. For a 3-by-3 neighborhood, there are 512 permutations, so the lookup table is a 512-element vector.

Once you create a lookup table, you can use it to perform the desired operation by using the applylut function.

The example below illustrates using lookup-table operations to modify an image containing text. You begin by writing a function that returns 1 if three or more pixels in the 3-by-3 neighborhood are 1; otherwise, it returns 0. You then call makelut, passing in this function as the first argument, and using the second argument to specify a 3-by-3 lookup table.

```
f = inline('sum(x(:)) >= 3');
lut = makelut(f,3);
```

lut is returned as a 512-element vector of 1's and 0's. Each value is the output from the function for one of the 512 possible permutations.

You then perform the operation using `applylut`.

```
BW1 = imread('text.tif');
BW2 = applylut(BW1,lut);
imshow(BW1)
figure, imshow(BW2)
```

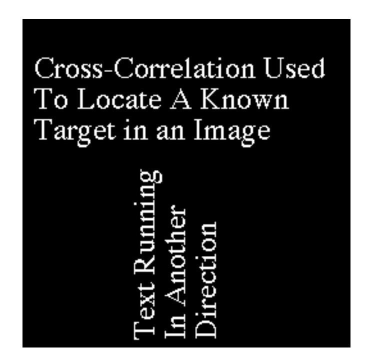




Figure 9-10: Text.tif Before and After Applying a Lookup Table Operation

For information about how `applylut` maps pixel combinations in the image to entries in the lookup table, see the reference page for `applylut`.

Note You cannot use `makelut` and `applylut` for neighborhoods of sizes other than 2-by-2 or 3-by-3. These functions support only 2-by-2 and 3-by-3 neighborhoods, because lookup tables are not practical for neighborhoods larger than 3-by-3. For example, a lookup table for a 4-by-4 neighborhood would have 65,536 entries.

10

Analyzing and Enhancing Images

This section describes the Image Processing Toolbox functions that support a range of standard image processing operations for analyzing and enhancing images. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Adaptive filter	A filter whose properties vary across an image depending on the local characteristics of the image pixels.
Contour	A path in an image along which the image intensity values are equal to a constant.
Edge	A curve that follows a path of rapid change in image intensity. Edges are often associated with the boundaries of objects in a scene. Edge detection is used to identify the edges in an image.
Property	A quantitative measurement of an image or image region. Examples of image region properties include centroid, bounding box, and area.
Histogram	A graph used in image analysis that shows the distribution of intensities in an image. The information in a histogram can be used to choose an appropriate enhancement operation. For example, if an image histogram shows that the range of intensity values is small, you can use an intensity adjustment function to spread the values across a wider range.
Noise	Errors in the image acquisition process that result in pixel values that do not reflect the true intensities of the real scene.
Profile	A set of intensity values taken from regularly spaced points along a line segment or multiline path in an image. For points that do not fall on the center of a pixel, the intensity values are interpolated.
Quadtree decomposition	An image analysis technique that partitions an image into homogeneous blocks.

Pixel Values and Statistics

The Image Processing Toolbox provides several functions that return information about the data values that make up an image. These functions return information about image data in various forms, including:

- The data values for selected pixels (`pixval`, `impixel`)
- The data values along a path in an image (`improfile`)
- A contour plot of the image data (`imcontour`)
- A histogram of the image data (`imhist`)
- Summary statistics for the image data (`mean2`, `std2`, `corr2`)
- Feature measurements for image regions (`imfeature`)

Pixel Selection

The toolbox includes two functions that provide information about the color data values of image pixels you specify:

- The `pixval` function interactively displays the data values for pixels as you move the cursor over the image. `pixval` can also display the Euclidean distance between two pixels.
- The `impixel` function returns the data values for a selected pixel or set of pixels. You can supply the coordinates of the pixels as input arguments, or you can select pixels using a mouse.

To use `pixval`, you first display an image and then enter the `pixval` command. `pixval` installs a black bar at the bottom of the figure, which displays the (x,y) coordinates for whatever pixel the cursor is currently over, and the color data for that pixel.

If you click on the image and hold down the mouse button while you move the cursor, `pixval` also displays the Euclidean distance between the point you clicked on and the current cursor location. `pixval` draws a line between these points to indicate the distance being measured. When you release the mouse button, the line and the distance display disappear.

`pixval` gives you more immediate results than `impixel`, but `impixel` has the advantage of returning its results in a variable, and it can be called either interactively or noninteractively. If you call `impixel` with no input arguments, the cursor changes to a crosshair when it is over the image. You can then click

on the pixels of interest; `impixel` displays a small star over each pixel you select. When you are done selecting pixels, press **Return**. `impixel` returns the color values for the selected pixels, and the stars disappear.

In this example, you call `impixel` and click on three points in the displayed image, and then press **Return**.

```
imshow canoe.tif
vals = impixel
```



```
vals =

0.1294    0.1294    0.1294
0.5176         0         0
0.7765    0.6118    0.4196
```

Notice that the second pixel, which is part of the canoe, is pure red; its green and blue values are both 0.

For indexed images, `pixval` and `impixel` both show the RGB values stored in the colormap, not the index values.

Intensity Profile

The `improfile` function calculates and plots the intensity values along a line segment or a multiline path in an image. You can supply the coordinates of the

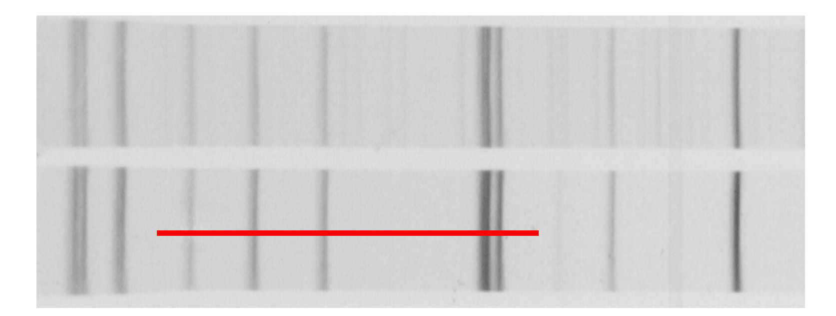
line segments as input arguments, or you can define the desired path using a mouse. In either case, `improfile` uses interpolation to determine the values of equally spaced points along the path. (By default, `improfile` uses nearest neighbor interpolation, but you can specify a different method. See Chapter 4, "Spatial Transformations", for a discussion of interpolation.) `improfile` works best with intensity and RGB images.

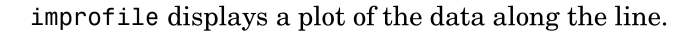
For a single line segment, `improfile` plots the intensity values in a two-dimensional view. For a multiline path, `improfile` plots the intensity values in a three-dimensional view.

If you call `improfile` with no arguments, the cursor changes to a cross hair when it is over the image. You can then specify line segments by clicking on the endpoints; `improfile` draws a line between each two consecutive points you select. When you finish specifying the path, press **Return**. `improfile` displays the plot in a new figure.

In this example, you call `improfile` and specify a single line with the mouse. The line is shown in red, and is drawn from left to right.

```
imshow debye1.tif
improfile
```





`improfile` displays a plot of the data along the line.

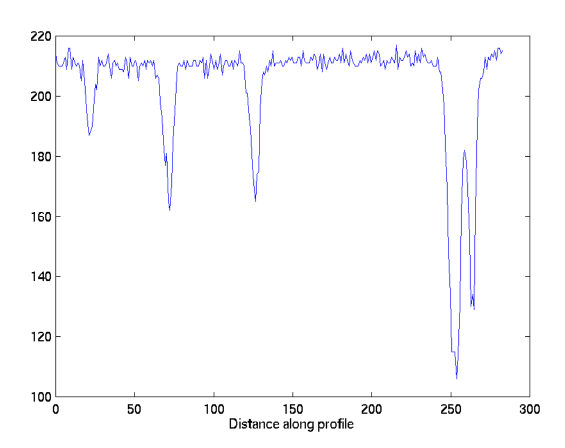


Figure 10-1: A Plot of Intensity Values Along a Line Segment in an Intensity Image

Notice the peaks and valleys and how they correspond to the light and dark bands in the image.

The example below shows how `improfile` works with an RGB image. The red line indicates where the line selection was made. Note that the line was drawn from top to bottom.

```
imshow flowers.tif
improfile
```



The `improfile` function displays a plot with separate lines for the red, green, and blue intensities.

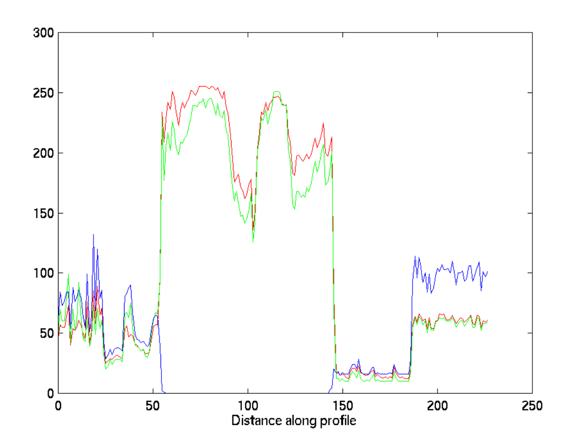


Figure 10-2: A Plot of Intensity Values Along a Line Segment in an RGB Image

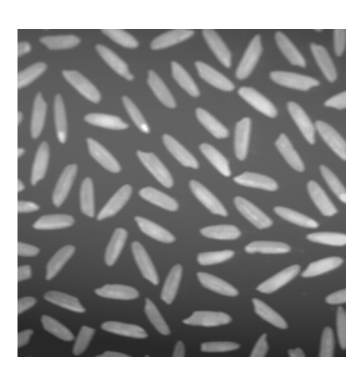
Notice how the lines correspond to the colors in the image. For example, the central region of the plot shows high intensities of green and red, while the blue intensity is 0. These are the values for the yellow flower.

Image Contours

You can use the toolbox function `imcontour` to display a contour plot of the data in an intensity image. This function is similar to the `contour` function in MATLAB, but it automatically sets up the axes so their orientation and aspect ratio match the image.

This example displays an intensity image of grains of rice and a contour plot of the image data.

```
I = imread('rice.tif');
imshow(I)
figure, imcontour(I)
```



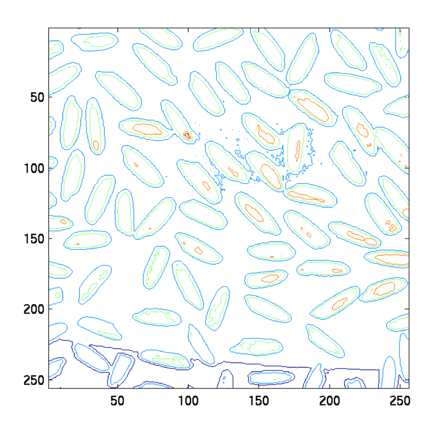


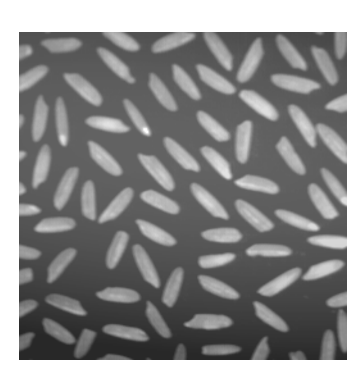
Figure 10-3: Rice.tif and Its Contour Plot

You can use the `clabel` function to label the levels of the contours. See the description of `clabel` in the MATLAB Function Reference for details.

Image Histogram

An *image histogram* is a chart that shows the distribution of intensities in an indexed or intensity image. The image histogram function `imhist` creates this plot by making `n` equally spaced bins, each representing a range of data values. It then calculates the number of pixels within each range. For example, the commands below display an image of grains of rice, and a histogram based on 64 bins.

```
I = imread('rice.tif');
imshow(I)
figure, imhist(I,64)
```



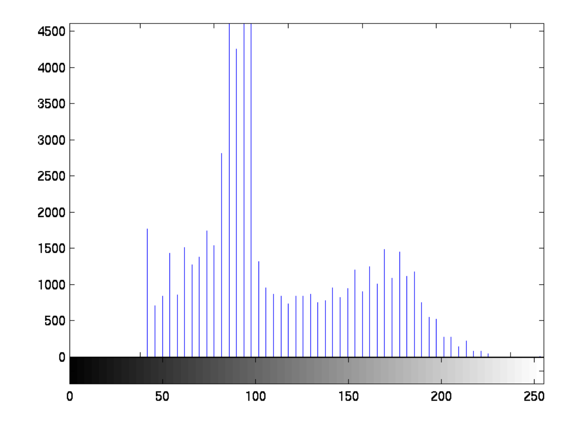


Figure 10-4: Rice.tif and Its Histogram

The histogram shows a peak at around 100, due to the dark gray background in the image. For information about how to modify an image by changing the distribution of its histogram, see "Intensity Adjustment" on page 10-14.

Summary Statistics

You can compute standard statistics of an image using the `mean2`, `std2`, and `corr2` functions. `mean2` and `std2` compute the mean and standard deviation of the elements of a matrix. `corr2` computes the correlation coefficient between two matrices of the same size.

These functions are two-dimensional versions of the `mean`, `std`, and `corrcoef` functions described in the MATLAB Function Reference.

Region Property Measurement

You can use the `regionprops` function to compute properties for image regions. For example, `regionprops` can measure such properties as the area, center of mass, and bounding box for a region you specify. See the reference page for `regionprops` for more information.

Image Analysis

Image analysis techniques return information about the structure of an image. This section describes toolbox functions that you can use for these image analysis techniques:

- Edge detection
- Quadtree decomposition

The functions described in this section work only with intensity images.

Edge Detection

You can use the `edge` function to detect edges, which are those places in an image that correspond to object boundaries. To find edges, this function looks for places in the image where the intensity changes rapidly, using one of these two criteria:

- Places where the first derivative of the intensity is larger in magnitude than some threshold
- Places where the second derivative of the intensity has a zero crossing

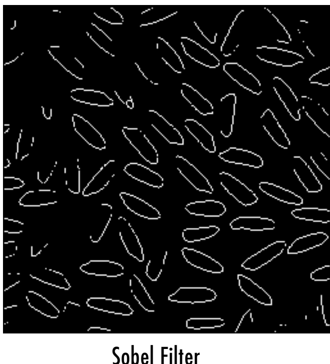
`edge` provides a number of derivative estimators, each of which implements one of the definitions above. For some of these estimators, you can specify whether the operation should be sensitive to horizontal or vertical edges, or both. `edge` returns a binary image containing 1's where edges are found and 0's elsewhere.

The most powerful edge-detection method that `edge` provides is the Canny method. The Canny method differs from the other edge-detection methods in that it uses two different thresholds (to detect strong and weak edges), and includes the weak edges in the output only if they are connected to strong edges. This method is therefore less likely than the others to be "fooled" by noise, and more likely to detect true weak edges.

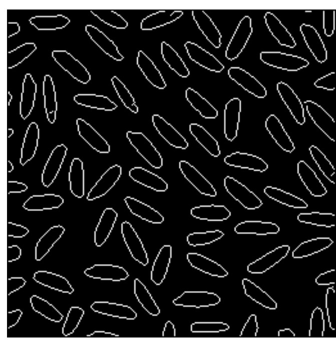
The example below illustrates the power of the Canny edge detector. It shows the results of applying the Sobel and Canny edge detectors to the `rice.tif` image.

```
I = imread('rice.tif');
BW1 = edge(I,'sobel');
BW2 = edge(I,'canny');
imshow(BW1)
```

```
figure, imshow(BW2)
```



Sobel Filter



Canny Filter

For an interactive demonstration of edge detection, try running `edgedemo`.

Quadtree Decomposition

Quadtree decomposition is an analysis technique that involves subdividing an image into blocks that are more homogeneous than the image itself. This technique reveals information about the structure of the image. It is also useful as the first step in adaptive compression algorithms.

You can perform quadtree decomposition using the `qtdecomp` function. This function works by dividing a square image into four equal-sized square blocks, and then testing each block to see if it meets some criterion of homogeneity (e.g., if all of the pixels in the block are within a specific dynamic range). If a block meets the criterion, it is not divided any further. If it does not meet the criterion, it is subdivided again into four blocks, and the test criterion is applied to those blocks. This process is repeated iteratively until each block meets the criterion. The result may have blocks of several different sizes.

For example, suppose you want to perform quadtree decomposition on a 128-by-128 intensity image. The first step is to divide the image into four 64-by-64 blocks. You then apply the test criterion to each block; for example, the criterion might be

```
max(block(:))   min(block(:)) <= 0.2
```

If one of the blocks meets this criterion, it is not divided any further; it is 64-by-64 in the final decomposition. If a block does not meet the criterion, it is

then divided into four 32-by-32 blocks, and the test is then applied to each of these blocks. The blocks that fail to meet the criterion are then divided into four 16-by-16 blocks, and so on, until all blocks "pass." Some of the blocks may be as small as 1-by-1, unless you specify otherwise.

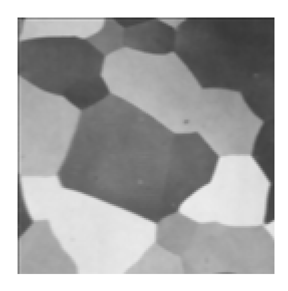
The call to `qtdecomp` for this example would be

```
S = qtdecomp(I,0.2)
```

`S` is returned as a sparse matrix whose nonzero elements represent the upper-left corners of the blocks; the value of each nonzero element indicates the block size. `S` is the same size as `I`.

Note The threshold value is specified as a value between 0 and 1, regardless of the class of `I`. If `I` is `uint8`, the threshold value you supply is multiplied by 255 to determine the actual threshold to use; if `I` is `uint16`, the threshold value you supply is multiplied by 65535.

The example below shows an image and a representation of its quadtree decomposition. Each black square represents a homogeneous block, and the white lines represent the boundaries between blocks. Notice how the blocks are smaller in areas corresponding to large changes in intensity in the image.



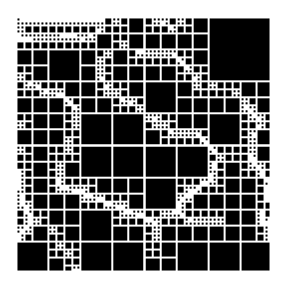


Figure 10-5: An Image (left) and a Representation of its Quadtree Decomposition

You can also supply `qtdecomp` with a function (rather than a threshold value) for deciding whether to split blocks; for example, you might base the decision on the variance of the block. See the reference page for `qtdecomp` for more information.

For an interactive demonstration of quadtree decomposition, try running `qtdemo`.

Image Enhancement

Image enhancement techniques are used to improve an image, where "improve" is sometimes defined objectively (e.g., increase the signal-to-noise ratio), and sometimes subjectively (e.g., make certain features easier to see by modifying the colors or intensities).

This section discusses these image enhancement techniques:

- "Intensity Adjustment"
- "Noise Removal"

The functions described in this section apply primarily to intensity images. However, some of these functions can be applied to color images as well. For information about how these functions work with color images, see the reference pages for the individual functions.

Intensity Adjustment

Intensity adjustment is a technique for mapping an image's intensity values to a new range. For example, `rice.tif.` is a low contrast image. The histogram of `rice.tif`, shown in Figure 10-4, indicates that there are no values below 40 or above 225. If you remap the data values to fill the entire intensity range [0, 255], you can increase the contrast of the image.

You can do this kind of adjustment with the `imadjust` function. The general syntax of `imadjust` is

```
J = imadjust(I,[low_in high_in],[low_out high_out])
```

where `low_in` and `high_in` are the intensities in the input image which are mapped to `low_out` and `high_out` in the output image. For example, this code performs the adjustment described above.

```
I = imread('rice.tif');
J = imadjust(I,[0.15 0.9],[0 1]);
```

The first vector passed to `imadjust`, `[0.15 0.9]`, specifies the low and high intensity values that you want to map. The second vector, `[0 1]`, specifies the scale over which you want to map them. Thus, the example maps the intensity value 0.15 in the input image to 0 in the output image, and 0.9 to 1.

Note that you must specify the intensities as values between 0 and 1 regardless of the class of I. If I is uint8, the values you supply are multiplied by 255 to determine the actual values to use; if I is uint16, the values are multiplied by 65535. To learn about an alternative way to set this limits automatically, see “Setting the Adjustment Limits Automatically” on page 10-16.

This figure displays the adjusted image and its histogram. Notice the increased contrast in the image, and that the histogram now fills the entire range.

```
imshow(J)
figure, imhist(J,64)
```



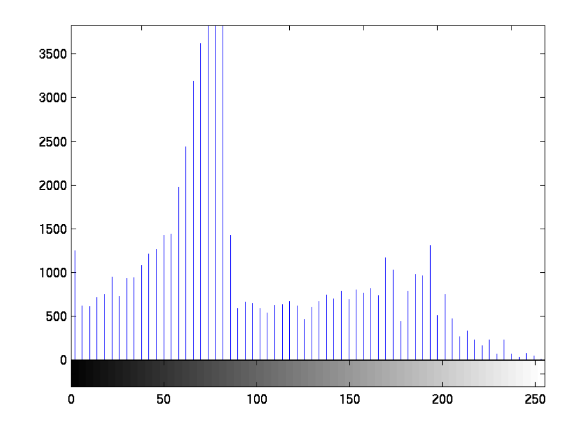


Figure 10-6: Rice.tif After an Intensity Adjustment and a Histogram of Its Adjusted Intensities

Similarly, you can decrease the contrast of an image by narrowing the range of the data, as in this call.

```
J = imadjust(I,[0 1],[0.3 0.8]);
```

In addition to increasing or decreasing contrast, you can perform a wide variety of other image enhancements with imadjust. In the example below, the man’s coat is too dark to reveal any detail. The call to imadjust maps the range [0,51] in the uint8 input image to [128,255] in the output image. This brightens the image considerably, and also widens the dynamic range of the dark portions of the original image, making it much easier to see the details in the coat. Note, however, that because all values above 51 in the original image get mapped to 255 (white) in the adjusted image, the adjusted image appears “washed out.”

```
I = imread('cameraman.tif');
J = imadjust(I,[0 0.2],[0.5 1]);
imshow(I)
figure, imshow(J)
```





Figure 10-7: Remapping and Widening the Dynamic Range

Setting the Adjustment Limits Automatically

To use `imadjust`, you must typically perform two steps:

1. View the histogram of the image to determine the intensity value limits.
2. Specify these limits as a fraction between 0.0 and 1.0 so that you can pass them to `imadjust` in the `[low_in high_in]` vector.

For a more convenient way to specify these limits, use the `stretchlim` function. This function calculates the histogram of the image and determines the adjustment limits automatically. The `stretchlim` function returns these values as fractions in a vector that you can pass as the `[low_in high_in]` argument to `imadjust`; for example,

```
I = imread('rice.tif');
J = imadjust(I,stretchlim(I),[0 1]);
```

By default, `stretchlim` uses the intensity values that represent the bottom 1% (0.01) and the top 1% (0.99) of the range as the adjustment limits. By trimming the extremes at both ends of the intensity range, `stretchlim` makes more room in the adjusted dynamic range for the remaining intensities. But you can

specify other range limits as an argument to `stretchlim`. See the `stretchlim` reference page for more information.

Gamma Correction

`imadjust` maps `low` to `bottom`, and `high` to `top`. By default, the values between `low` and `high` are mapped linearly to values between `bottom` and `top`. For example, the value halfway between `low` and `high` corresponds to the value halfway between `bottom` and `top`.

`imadjust` can accept an additional argument which specifies the *gamma correction* factor. Depending on the value of gamma, the mapping between values in the input and output images may be nonlinear. For example, the value halfway between `low` and `high` may map to a value either greater than or less than the value halfway between `bottom` and `top`.

Gamma can be any value between 0 and infinity. If gamma is 1 (the default), the mapping is linear. If gamma is less than 1, the mapping is weighted toward higher (brighter) output values. If gamma is greater than 1, the mapping is weighted toward lower (darker) output values.

The figure below illustrates this relationship. The three transformation curves show how values are mapped when gamma is less than, equal to, and greater than 1. (In each graph, the *x*-axis represents the intensity values in the input image, and the *y*-axis represents the intensity values in the output image.)

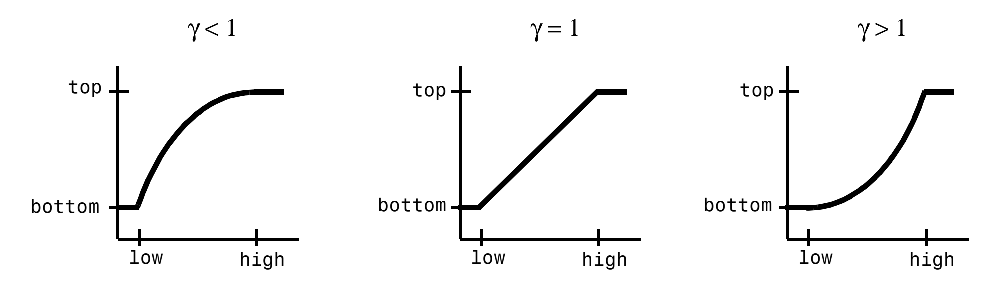


Figure 10-8: Plots Showing Three Different Gamma Correction Settings

The example below illustrates gamma correction. Notice that in the call to `imadjust`, the data ranges of the input and output images are specified as empty matrices. When you specify an empty matrix, `imadjust` uses the default range of [0,1]. In the example, both ranges are left empty; this means that gamma correction is applied without any other adjustment of the data.

```
[X,map] = imread('forest.tif')
I = ind2gray(X,map);
J = imadjust(I,[],[],0.5);
imshow(I)
figure, imshow(J)
```





Figure 10-9: Forest.tif Before and After Applying Gamma Correction of 0.5

Histogram Equalization

The process of adjusting intensity values can be done automatically by the `histeq` function. `histeq` performs *histogram equalization,* which involves transforming the intensity values so that the histogram of the output image approximately matches a specified histogram. (By default, `histeq` tries to match a flat histogram with 64 bins, but you can specify a different histogram instead; see the reference page for `histeq`.)

This example illustrates using `histeq` to adjust an intensity image. The original image has low contrast, with most values in the middle of the intensity range. `histeq` produces an output image having values evenly distributed throughout the range.

```
I = imread('pout.tif');
J = histeq(I);
imshow(I)
figure, imshow(J)
```

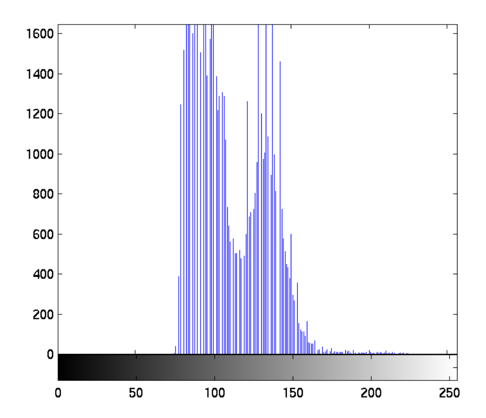




Figure 10-10: Pout.tif Before and After Histogram Equalization

The example below shows the histograms for the two images.

```
figure, imhist(I)
figure, imhist(J)
```



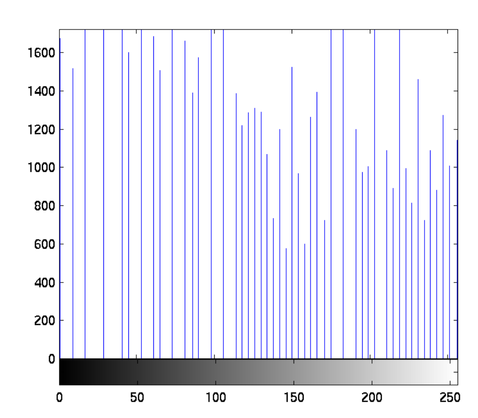
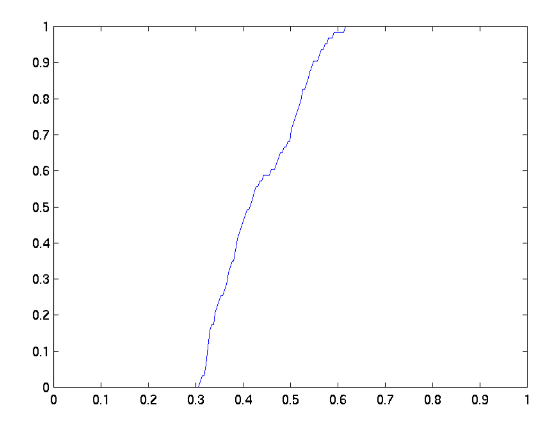


Figure 10-11: Histogram Before Equalization (left) and After Equalization (right)

`histeq` can return an additional 1-by-256 vector that shows, for each possible input value, the resulting output value. (The values in this vector are in the

range [0,1], regardless of the class of the input image.) You can plot this data to get the transformation curve. For example,

```
I = imread('pout.tif');
[J,T] = histeq(I);
figure,plot((0:255)/255,T);
```



Notice how this curve reflects the histograms in the previous figure, with the input values being mostly between 0.3 and 0.6, while the output values are distributed evenly between 0 and 1.

For an interactive demonstration of intensity adjustment, try running `imadjdemo`.

Noise Removal

Digital images are prone to a variety of types of noise. There are several ways that noise can be introduced into an image, depending on how the image is created. For example:

- If the image is scanned from a photograph made on film, the film grain is a source of noise. Noise can also be the result of damage to the film, or be introduced by the scanner itself.
- If the image is acquired directly in a digital format, the mechanism for gathering the data (such as a CCD detector) can introduce noise.

- Electronic transmission of image data can introduce noise.

The toolbox provides a number of different ways to remove or reduce noise in an image. Different methods are better for different kinds of noise. The methods available include:

- Linear filtering
- Median filtering
- Adaptive filtering

Also, in order to simulate the effects of some of the problems listed above, the toolbox provides the `imnoise` function, which you can use to *add* various types of noise to an image. The examples in this section use this function.

Linear Filtering

You can use linear filtering to remove certain types of noise. Certain filters, such as averaging or Gaussian filters, are appropriate for this purpose. For example, an averaging filter is useful for removing grain noise from a photograph. Because each pixel gets set to the average of the pixels in its neighborhood, local variations caused by grain are reduced.

See “Linear Filtering” on page 7-4 for more information.

Median Filtering

Median filtering is similar to using an averaging filter, in that each output pixel is set to an “average” of the pixel values in the neighborhood of the corresponding input pixel. However, with median filtering, the value of an output pixel is determined by the *median* of the neighborhood pixels, rather than the mean. The median is much less sensitive than the mean to extreme values (called *outliers*). Median filtering is therefore better able to remove these outliers without reducing the sharpness of the image.

The `medfilt2` function implements median filtering. The example below compares using an averaging filter and `medfilt2` to remove *salt and pepper* noise. This type of noise consists of random pixels being set to black or white (the extremes of the data range). In both cases the size of the neighborhood used for filtering is 3-by-3.

First, read in the image and add noise to it.

```
I = imread('eight.tif');
```

```
J = imnoise(I,'salt & pepper',0.02);
imshow(I)
figure, imshow(J)
```



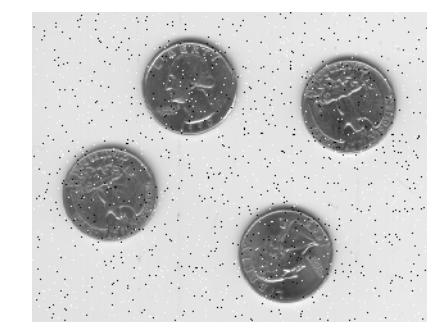
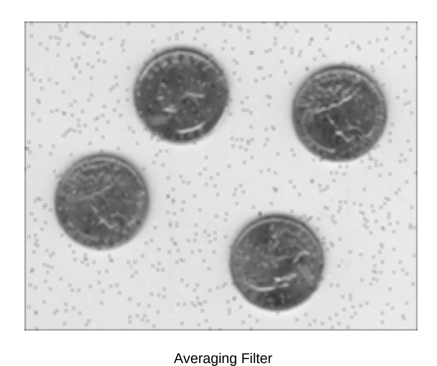


Figure 10-12: Eight.tif Before and After Adding Salt-and-Pepper Noise

Now filter the noisy image and display the results. Notice that `medfilt2` does a better job of removing noise, with less blurring of edges.

```
K = filter2(fspecial('average',3),J)/255;
L = medfilt2(J,[3 3]);
figure, imshow(K)
figure, imshow(L)
```





Averaging Filter

Figure 10-13: Noisy Version of Eight.tif Filtered with Averaging Filter (left) and Median Filter (right)

Median filtering is a specific case of *order-statistic filtering*, also known as *rank filtering*. For information about order-statistic filtering, see the reference page for the `ordfilt2` function.

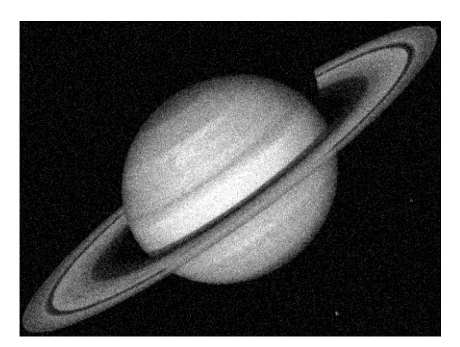
Adaptive Filtering

The `wiener2` function applies a Wiener filter (a type of linear filter) to an image *adaptively,* tailoring itself to the local image variance. Where the variance is large, `wiener2` performs little smoothing. Where the variance is small, `wiener2` performs more smoothing.

This approach often produces better results than linear filtering. The adaptive filter is more selective than a comparable linear filter, preserving edges and other high frequency parts of an image. In addition, there are no design tasks; the `wiener2` function handles all preliminary computations, and implements the filter for an input image. `wiener2`, however, does require more computation time than linear filtering.

`wiener2` works best when the noise is constant-power ("white") additive noise, such as Gaussian noise. The example below applies `wiener2` to an image of Saturn that has had Gaussian noise added.

```
I = imread('saturn.tif');
J = imnoise(I,'gaussian',0,0.005);
K = wiener2(J,[5 5]);
imshow(J)
figure, imshow(K)
```



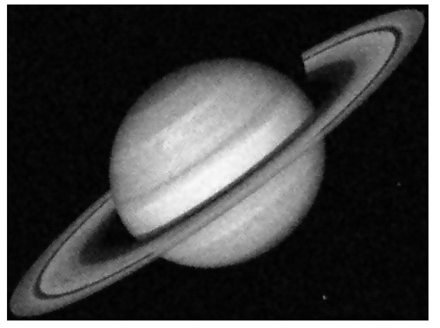


Figure 10-14: Noisy Version of Saturn.tif Before and After Adaptive Filtering

For an interactive demonstration of filtering to remove noise, try running `nrfiltdemo`.

11

Region-Based Processing

This section describes operations that you can perform on a selected region of an image. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Binary mask	A binary image with the same size as the image you want to process. The mask contains 1's for all pixels that are part of the region of interest, and 0's everywhere else.
Filling a region	A process that "fills" a region of interest by interpolating the pixel values from the borders of the region. This process can be used to make objects in an image seem to disappear as they are replaced with values that blend in with the background area.
Filtering a region	The process of applying a filter to a region of interest. For example, you can apply an intensity adjustment filter to certain regions of an image.
Interpolation	The process by which we estimate an image value at a location in between image pixels.
Masked filtering	An operation that applies filtering only to the regions of interest in an image that are identified by a binary mask. Filtered values are returned for pixels where the binary mask contains 1's; unfiltered values are returned for pixels where the binary mask contains 0's.
Region of interest	A portion of an image that you want to filter or perform some other operation on. You define a region of interest by creating a binary mask. There can be more than one region defined in an image. The regions can be "geographic" in nature, such as polygons that encompass contiguous pixels, or they can be defined by a range of intensities. In the latter case, the pixels are not necessarily contiguous.

Specifying a Region of Interest

A *region of interest* is a portion of an image that you want to filter or perform some other operation on. You define a region of interest by creating a *binary mask*, which is a binary image with the same size as the image you want to process. The mask contains 1's for all pixels that are part of the region of interest, and 0's everywhere else.

The following subsections discuss methods for creating binary masks:

- "Selecting a Polygon" on page 11-3
- "Other Selection Methods" on page 11-4 (using any binary mask or the `roicolor` function)

For an interactive demonstration of region-based processing, try running `roidemo`.

Selecting a Polygon

You can use the `roipoly` function to specify a polygonal region of interest. If you call `roipoly` with no input arguments, the cursor changes to a cross hair when it is over the image displayed in the current axes. You can then specify the vertices of the polygon by clicking on points in the image with the mouse. When you are done selecting vertices, press **Return**; `roipoly` returns a binary image of the same size as the input image, containing 1's inside the specified polygon, and 0's everywhere else.

The example below illustrates using the interactive syntax of `roipoly` to create a binary mask. The border of the selected region in Figure 11-1, which was created using a mouse, is shown in red.

```
I = imread('pout.tif');
imshow(I)
BW = roipoly;
```



Figure 11-1: A Polygonal Region of Interest Selected Using roipoly

```
imshow(BW)
```

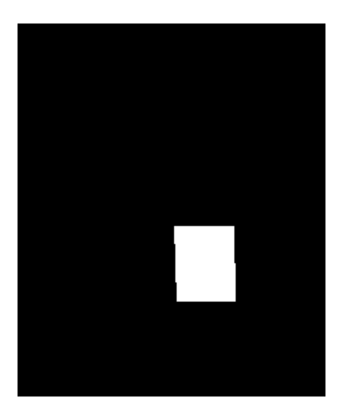


Figure 11-2: A Binary Mask Created for the Region Shown in Figure 11-1.

You can also use `roipoly` noninteractively. See the reference page for `roipoly` for more information.

Other Selection Methods

`roipoly` provides an easy way to create a binary mask. However, you can use *any* binary image as a mask, provided that the binary image is the same size as the image being filtered.

For example, suppose you want to filter the intensity image `I`, filtering only those pixels whose values are greater than 0.5. You can create the appropriate mask with this command.

```
BW = (I > 0.5);
```

You can also use the `roicolor` function to define the region of interest based on a color or intensity range. For more information, see the reference page for `roicolor`.

Filtering a Region

You can use the `roifilt2` function to process a region of interest. When you call `roifilt2`, you specify an intensity image, a binary mask, and a filter. `roifilt2` filters the input image and returns an image that consists of filtered values for pixels where the binary mask contains 1's, and unfiltered values for pixels where the binary mask contains 0's. This type of operation is called *masked filtering*.

This example uses the mask created in the example in "Selecting a Polygon" on page 11-3 to increase the contrast of the logo on the girl's coat.

```
h = fspecial('unsharp');
I2 = roifilt2(h,I,BW);
imshow(I)
figure, imshow(I2)
```





Figure 11-3: An Image Before and After Using an Unsharp Filter on the Region of Interest.

`roifilt2` also enables you to specify your own function to operate on the region of interest. In the example below, the `imadjust` function is used to lighten parts of an image. The mask in the example is a binary image containing text. The resulting image has the text imprinted on it.

```
BW = imread('text.tif');
I = imread('cameraman.tif');
f = inline('imadjust(x,[],[],0.3)');
I2 = roifilt2(I,BW,f);
```

```
imshow(I2)
```

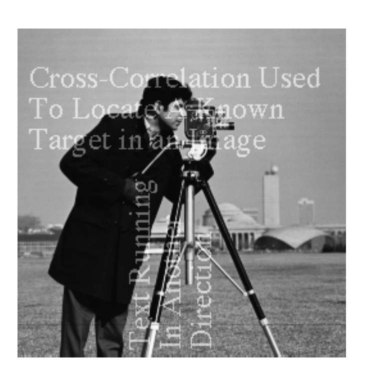


Figure 11-4: An Image Brightened Using a Binary Mask Containing Text

Note that `roifilt2` is best suited to operations that return data in the same range as in the original image because the output image takes some of its data directly from the input image. Certain filtering operations can result in values outside the normal image data range (i.e., [0,1] for images of class `double`, [0,255] for images of class `uint8`, [0,65535] for images of class `uint16`). For more information, see the reference page for `roifilt2`.

Filling a Region

You can use the `roifill` function to fill a region of interest, interpolating from the borders of the region. This function is useful for image editing, including removal of extraneous details or artifacts.

`roifill` performs the fill operation using an interpolation method based on Laplace's equation. This method results in the smoothest possible fill, given the values on the boundary of the region.

As with `roipoly`, you select the region of interest with the mouse. When you complete the selection, `roifill` returns an image with the selected region filled in.

This example uses `roifill` to modify the `trees` image. The border of the selected region is shown in red on the original image.

```
load trees
I = ind2gray(X,map);
imshow(I)
I2 = roifill;
```



Figure 11-5: A Region of Interest Selected for Filling

```
imshow(I2)
```



Figure 11-6: The Region of Interest Shown in Figure 11-5 Has Been Filled

12

Image Deblurring

This section describes how to deblur an image using the toolbox deblurring functions. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	MATLAB Definition
Deconvolution	The process of reversing the effect of convolution.
Distortion operator	The operator that describes a process causing the acquired image to be different from the original scene. Distortion caused by a Point Spread Function (see below) is just one type of distortion.
Optical transfer function (OTF)	In the frequency domain, the OTF describes the response of a linear, position invariant system to an impulse. The OTF is the Fourier transform of the point spread function (PSF).
Point spread function (PSF)	In the spatial domain, the PSF describes the degree to which an optical system blurs (spreads) a point of light. The PSF is the inverse Fourier transform of the OTF.

Understanding Deblurring

This section provides some background on deblurring techniques. The section includes these topics:

- "Causes of Blurring"
- "Deblurring Model"

Causes of Blurring

The blurring, or degradation, of an image can be caused by many factors:

- Movement during the image capture process, by the camera or, when long exposure times are used, by the subject
- Out-of-focus optics, use of a wide-angle lens, atmospheric turbulence, or a short exposure time, which reduced the number of photons captured
- Scattered light distortion in confocal microscopy

Deblurring Model

A blurred or degraded image can be approximately described by this equation **g = Hf + n**, where:

g = The blurred image

H = The distortion operator, also called the *point-spread function* (PSF). This function, when convolved with the image, creates the distortion

f = The original true image

n = Additive noise, introduced during image acquisition, that corrupts the image

---

Note The image **f** really doesn't exist. This image represents what you would have if you had perfect image acquisition conditions.

The Importance of the PSF

Based on this model, the fundamental task of deblurring is to deconvolve the blurred image with the PSF that exactly describes the distortion.

Note The quality of the deblurred image is mainly determined by knowledge of the PSF.

To illustrate, this example takes a clear image and deliberately blurs it by convolving it with a PSF. The example uses the `fspecial` function to create a PSF that simulates a motion blur, specifying the length of the blur in pixels, (`LEN=31`), and the angle of the blur in degrees (`THETA=11`). Once the PSF is created, the example uses the `imfilter` function to convolve the PSF with the original image, `I`, to create the blurred image, `Blurred`. (To see how deblurring is the reverse of this process, using the same images, see "Deblurring with the Wiener Filter" on page 12-6.)

```
I = imread('flowers.tif');
I = I(10+[1:256],222+[1:256],:); % crop the image
figure;imshow(I);title('Original Image');

LEN = 31;
THETA = 11;
PSF = fspecial('motion',LEN,THETA);
Blurred = imfilter(I,PSF,'circular','conv');
figure; imshow(Blurred);title('Blurred Image');
```





Original image



Blurred image

Using the Deblurring Functions

The toolbox includes four deblurring functions, listed here in order of complexity:

- `deconvwnr` — Implements deblurring using the Wiener filter
- `deconvreg` — Implements deblurring using a regularized filter
- `deconvlucy` — Implements deblurring using the Lucy-Richardson algorithm
- `deconvblind` — Implements deblurring using the blind deconvolution algorithm

All the functions accept a PSF and the blurred image as their primary arguments. The `deconvwnr` function implements a least squares solution. The `deconvreg` function implements a constrained least squares solution, where you can place constraints on the output image (the smoothness requirement is the default). With either of these functions, you should provide some information about the noise to reduce possible noise amplification during deblurring.

The `deconvlucy` function implements an accelerated, damped Lucy-Richardson algorithm. This function performs multiple iterations, using optimization techniques and Poisson statistics. With this function, you do not need to provide information about the additive noise in the corrupted image.

The `deconvblind` function implements the blind deconvolution algorithm, which performs deblurring without knowledge of the PSF. When you call `deconvblind`, you pass as an argument your initial guess at the PSF. The `deconvblind` function returns a restored PSF in addition to the restored image. The implementation uses the same damping and iterative model as the `deconvlucy` function.

Note You may need to perform many iterations of the deblurring process, varying the parameters you specify to the deblurring functions with each iteration, until you achieve an image that, based on the limits of your information, is the best approximation of the original scene. Along the way, you must make numerous judgements about whether newly uncovered features in the image are features of the original scene, or simply artifacts of the deblurring process.

For information about creating your own deblurring functions, see "Creating Your Own Deblurring Functions" on page 12-19. To avoid "ringing" in a deblurred image, you can use the `edgetaper` function to preprocess your image before passing it to the deblurring functions. See "Avoiding Ringing in Deblurred Images" on page 12-20 for more information.

Deblurring with the Wiener Filter

Use the `deconvwnr` function to deblur an image using the Wiener filter. Wiener deconvolution can be used effectively when the frequency characteristics of the image and additive noise are known, to at least some degree. In the absence of noise, the Wiener filter reduces to the ideal inverse filter.

This example deblurs the blurred flower image, created in "Deblurring Model" on page 12-3, specifying the same PSF function that was used to create the blur.

```
I = imread('flowers.tif');
I = I(10+[1:256],222+[1:256],:); % crop the image
figure;imshow(I);title('Original Image');

% create PSF
LEN = 31;
THETA = 11;
PSF = fspecial('motion',LEN,THETA);

% blur the image
Blurred = imfilter(I,PSF,'circular','conv');
figure; imshow(Blurred);title('Blurred Image');

% deblur the image
wnr1 = deconvwnr(Blurred,PSF);
figure;imshow(wnr1);
title('Restored, True PSF');
```


Original image


Blurred image


Image restored by Wiener filter

This example illustrates the importance of knowing the PSF, the function that caused the blur. When you know the exact PSF, as in this example, the results of deblurring can be quite effective.

Refining the Result

You can affect the deconvolution results by providing values for the optional arguments supported by the `deconvwnr` function. Using these arguments you can specify the noise-to-signal power value and/or provide autocorrelation functions to help refine the result of deblurring. To see the impact of these optional arguments, view the Image Processing Toolbox Deblurring Demos.

Deblurring with a Regularized Filter

Use the deconvreg function to deblur an image using a regularized filter. A regularized filter can be used effectively when limited information is known about the additive noise.

To illustrate, this example simulates a blurred image by convolving a Gaussian filter PSF with an image (using imfilter). Additive noise in the image is simulated by adding Gaussian noise of variance V to the blurred image (using imnoise).

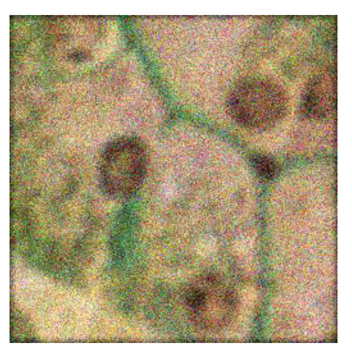
```
I = imread('tissue1.tif');
I = I(350+[1:256],1:256,:);
figure;imshow(I);title('Original Image');

% Create the PSF
PSF = fspecial('gaussian',11,5);

% Blur the image
Blurred = imfilter(I,PSF,'conv');

% Add noise
V = .02;
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);
figure;imshow(BlurredNoisy);title('Blurred & Noisy');
```





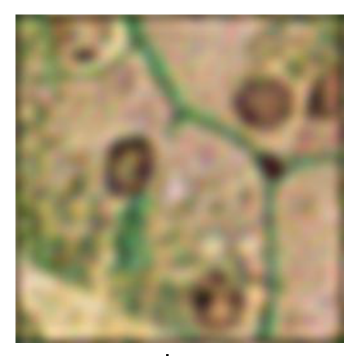


Original image

Blurred & noisy image

Use deconvreg to deblur the image, specifying the PSF used to create the blur, and the noise-to-signal power ratio, NP.

```
NP = V*prod(size(I)); % noise power
[reg1 LAGRA] = deconvreg(BlurredNoisy,PSF,NP);
figure,imshow(reg1),title('Restored with NP');
```



Restored image

Refining the Result

You can affect the deconvolution results by providing values for the optional arguments supported by the `deconvreg` function. Using these arguments you can specify the noise power value, the range over which `deconvreg` should iterate as it converges on the optimal solution, and the regularization operator to constrain the deconvolution. To see the impact of these optional arguments, view the Image Processing Toolbox Deblurring Demos.

Deblurring with the Lucy-Richardson Algorithm

Use the `deconvlucy` function to deblur an image using the accelerated, damped, Lucy-Richardson algorithm. The algorithm maximizes the likelihood that the resulting image, when convolved with the PSF, is an instance of the blurred image, assuming Poisson noise statistics. This function can be effective when you know the PSF, but know little about the additive noise in the image.

The `deconvlucy` function implements several adaptations to the original Lucy-Richardson maximum likelihood algorithm which address complex image restoration tasks. Using these adaptations, you can:

- Reduce the effect of noise amplification on image restoration
- Account for nonuniform image quality (e.g., bad pixels, flat-field variation)
- Handle camera read-out and background noise

- Improve the restored image resolution by subsampling

The following sections provide more information about each of these adaptations.

Reducing the Effect of Noise Amplification

Noise amplification is a common problem of maximum likelihood methods that attempt to fit data as closely as possible. After many iterations, the restored image can have a speckled appearance, especially for a smooth object observed at low signal-to-noise ratios. These speckles do not represent any real structure in the image, but are artifacts of fitting the noise in the image too closely.

To control noise amplification, the `deconvlucy` function uses a damping parameter, `DAMPAR`. This parameter specifies the threshold level for the deviation of the resulting image from the original image, below which damping occurs. For pixels that deviate in the vicinity of their original value, iterations are suppressed.

Damping is also used to reduce *ringing*, the appearance of high-frequency structures in a restored image. Ringing is not necessarily the result of noise amplification. See “Avoiding Ringing in Deblurred Images” on page 12-20 for more information.

Accounting for Nonuniform Image Quality

Another complication of real-life image restoration is that the data might include bad pixels, or that the quality of the receiving pixels might vary with time and position. By specifying the `WEIGHT` array parameter with the `deconvlucy` function, you can specify that certain pixels in the image be ignored. To ignore a pixel, assign a weight of zero to the element in the `WEIGHT` array that corresponds to the pixel in the image.

The algorithm converges on predicted values for the bad pixels based on the information from neighborhood pixels. The variation in the detector response from pixel to pixel (the so-called flat-field correction) can also be accommodated by the `WEIGHT` array. Instead of assigning a weight of 1.0 to the good pixels, you can specify fractional values and weight the pixels according to the amount of the flat-field correction.

Handling Camera Read-Out Noise

Noise in charge coupled device (CCD) detectors has two primary components:

- Photon counting noise with a Poisson distribution
- Read-out noise with a Gaussian distribution

The Lucy-Richardson iterations intrinsically account for the first type of noise. You must account for the second type of noise; otherwise, it may cause pixels with low levels of incident photons to have negative values.

The `deconvlucy` function uses the `READOUT` input parameter to handle camera read-out noise. The value of this parameter is typically the sum of the read-out noise variance and the background noise (e.g., number of counts from the background radiation.) The value of the `READOUT` parameter specifies an offset that ensures that all values are positive.

Handling Undersampled Images

The restoration of undersampled data can be improved significantly if it is done on a finer grid. The `deconvlucy` function uses the `SUBSMPL` parameter to specify the subsampling rate, if the PSF is known to have a higher resolution.

If the undersampled data is the result of camera pixel binning during image acquisition, the PSF observed at each pixel rate would serve as a finer grid PSF. Otherwise, the PSF could be obtained via observations taken at subpixel offsets or via optical modeling techniques. This method is especially effective for the images of stars (high signal-to-noise ratio), because the stars are effectively forced to be in the center of a pixel. If a star is centered between the pixels, it will be restored as a combination of the neighboring pixels. A finer grid would redirect the consequent spreading of the star flux back to the center of the star's image.

Creating a Sample Blurred Image

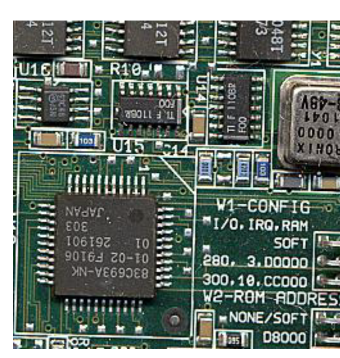
To illustrate a simple use of `deconvlucy`, this example simulates a blurred, noisy image by convolving a Gaussian filter PSF with an image (using `imfilter`) and then adding Gaussian noise of variance `V` to the blurred image (using `imnoise`).

```
I = imread('board.tif');
I = I(50+[1:256],2+[1:256],:);
figure;imshow(I);title('Original Image');

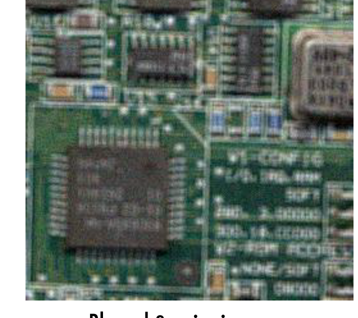
% Create the PSF
PSF = fspecial('gaussian',5,5);

% Simulate the blur
Blurred = imfilter(I,PSF,'symmetric','conv');

% Add noise
V = .002;
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);
figure;imshow(BlurredNoisy);title('Blurred & Noisy');
```



Original image



Blurred & noisy image

Deblurring the Image

Use `deconvlucy` to restore the blurred and noisy image, specifying the PSF used to create the blur, and limiting the number of iterations to 5 (the default is 10).

Note The `deconvlucy` function can return values in the output image that are beyond the range of the input image.

```
luc1 = deconvlucy(BlurredNoisy,PSF,5);
figure; imshow(luc1);
title('Restored Image, NUMIT = 5');
```



Restored image

Refining the Result

The `deconvlucy` function, by default, performs multiple iterations of the deblurring process. You can stop the processing, after a certain number of iterations, to check the result, and then restart the iterations from the point where processing stopped. To do this, pass in the input image as a cell array, for example, `{BlurredNoisy}`. The `deconvlucy` function returns the output image as a cell array which you can then pass as an input argument to `deconvlucy` to restart the deconvolution.

The output cell array contains these four elements:

- `output{1}` — The original input image
- `output{2}` — The image produced by the last iteration
- `output{3}` — The image produced by the next-to-last iteration
- `output{4}` — Internal information used by `deconvlucy` to know where to restart the process

The `deconvlucy` function supports several other optional arguments you can use to achieve the best possible result, such as specifying a damping parameter to handle additive noise in the blurred image. To see the impact of these optional arguments, view the Image Processing Toolbox Deblurring Demos.

Deblurring with the Blind Deconvolution Algorithm

Use the `deconvblind` function to deblur an image using the blind deconvolution algorithm. The algorithm maximizes the likelihood that the resulting image, when convolved with the resulting PSF, is an instance of the blurred image, assuming Poisson noise statistics. The blind deconvolution algorithm can be used effectively when no information about the distortion (blurring and noise) is known. The `deconvblind` function restores the image and the PSF simultaneously, using an iterative process similar to the accelerated, damped Lucy-Richardson algorithm.

The `deconvblind` function, just like the `deconvlucy` function, implements several adaptations to the original Lucy-Richardson maximum likelihood algorithm, which address complex image restoration tasks. Using these adaptations, you can:

- Reduce the effect of noise on the restoration
- Account for nonuniform image quality (e.g., bad pixels)
- Handle camera read-out noise

For more information about these adaptations, see "Deblurring with the Lucy-Richardson Algorithm" on page 12-9. In addition, the `deconvblind` function supports PSF constraints that can be passed in through a user-specified function.

Creating a Sample Blurred Image

To illustrate blind deconvolution, this example simulates a blurred image by convolving a motion filter PSF with an image (using `imfilter`).

```
I = imread('cameraman.tif');
figure; imshow(I); title('Original Image');

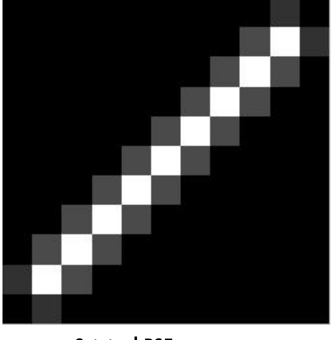
PSF = fspecial('motion',13,45);% Create the PSF
figure; imshow(PSF,[]); title('True PSF');

Blurred = imfilter(I,PSF,'circ','conv');% Simulate the blur
```

```
figure; imshow(Blurred); title('Blurred Image');
```



Original image



Original PSF



Blurred image

Image Restoration: First Pass

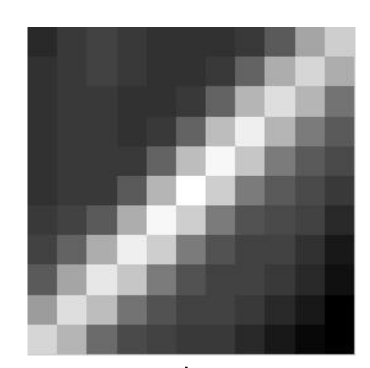
As a first pass at restoring the blurred image of the cameraman, call the `deconvblind` function specifying the image and an initial guess at the PSF as arguments.

When you specify the PSF, you must estimate its size and the values it contains. Your guess at the size of the initial PSF is more important to the ultimate success of the restoration than the values in the PSF. To determine the size, examine the blurred image and measure the width of a blur (in pixels) around an obviously sharp object. For example, in the sample blurred image, you can measure the blur near the contour of the man's sleeve. Because your initial guess at the values in the PSF is less important than the size, you can typically specify an array of 1's as the initial PSF.

The following example shows a restoration where the initial guess at the PSF is the same size as the true PSF that caused the blur.

```
INITPSF = ones(size(PSF));
[J P]= deconvblind(Blurred,INITPSF,30);
figure; imshow(J); title('Preliminary Restoration');
figure; imshow(P,[],'notruesize');
title('Preliminary Restoration');
```


Restored image


Restored PSF

This example specified an initial PSF that was the same size as the true PSF, i.e., the PSF used to create the blur. In a real application, you may need to rerun `deconvblind`, experimenting with PSFs of different sizes, until you achieve a satisfactory result. The restored PSF returned by each deconvolution can also provide valuable hints at the optimal PSF size. See the Image Processing Toolbox Deblurring Demos for an example.

Image Restoration: Second Pass

Although the first pass did succeed in deblurring the image to some extent, the ringing in the restored image around the sharp intensity contrast areas is unsatisfactory. (The example deliberately eliminated edge-related ringing by using the `'circular'` option in `imfilter` while creating a blurred image.) This example makes a second pass at deblurring, this time achieving a better result by using both the optional `WEIGHT` array parameter and by refining the guess at the initial PSF, `P1`.

Creating a WEIGHT Array. To reduce this contrast-related ringing, rerun the deconvolution, this time using the optional `WEIGHT` array parameter to exclude areas of high-contrast from the deblurring operation. You exclude a pixel from processing by assigning the value `0` to the corresponding element in the `WEIGHT` array. (See "Accounting for Nonuniform Image Quality" on page 12-10 for information about `WEIGHT` arrays.)

This example uses edge detection and morphological processing to create a `WEIGHT` array. The `edge`, `strel`, and `imdilate` functions detect the high-contrast areas in the image. Because the blur in the image is linear, the example dilates the image twice. (For more information about using these

functions, see Chapter 9, “Morphological Operations.”) To exclude the image boundary pixels (a high-contrast area) from processing, the example uses `padarray` to assign the value 0 to all border pixels.

```
WEIGHT = edge(I,'sobel',.28);
se1 = strel('disk',1);
se2 = strel('line',13,45);
WEIGHT = ~imdilate(WEIGHT,[se1 se2]);
WEIGHT = padarray(WEIGHT([1:2 end-[0:1]],[1:2 end-[0:1]]),[2 2]);
figure; imshow(WEIGHT); title('Weight array');
```



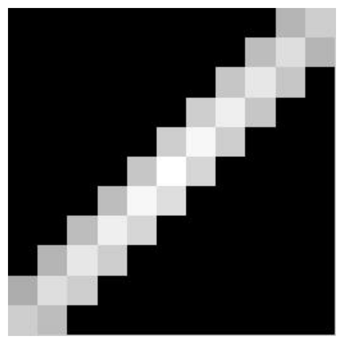
Weight array

Constraining the Restored PSF. Before repeating the deconvolution with the `WEIGHT` array, the example refines the guess at the PSF. The reconstructed PSF, `P`, returned by the first pass at deconvolution shows a clear linearity (see the image of the Restored PSF in “Image Restoration: First Pass” on page 12-15). For the second pass, the example uses a new PSF, `P1`, which is same as the restored PSF but with the small amplitude pixels set to 0.

```
P1 = P;
P1(find(P1 < 0.01))=0;
[J2 P2] = deconvblind(Blurred,P1,50,[],WEIGHT);
figure; imshow(J2);
title('Newly deblurred image');
figure; imshow(P2,[],'notruesize');
title('Newly reconstructed PSF');
```



Newly deblurred image



Newly reconstructed PSF

Refining the Result

The `deconvblind` function, by default, performs multiple iterations of the deblurring process. You can stop the processing, after a certain number of iterations, to check the result, and then restart the iterations from the point where processing stopped. To use this feature, you must pass in both the blurred image and the PSF as cell arrays, for example, `{Blurred}` and `{INITPSF}`.

The `deconvblind` function returns the output image and the restored PSF as cell arrays. The output image cell array contains these four elements:

`output{1}` — The original input image

`output{2}` — The image produced by the last iteration

`output{3}` — The image produced by the next-to-last iteration

`output{4}` — Internal information used by `deconvblind` to know where to restart the process

The PSF output cell array contains similar elements.

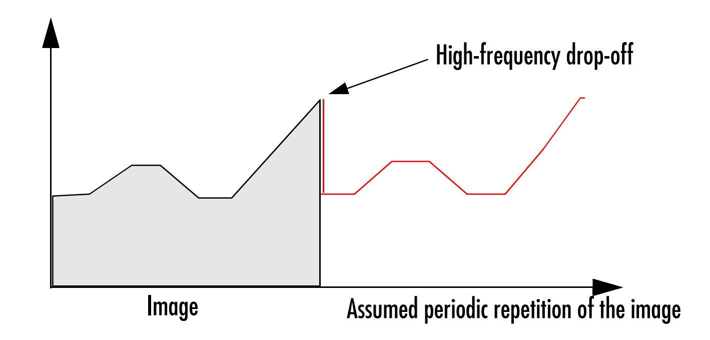
The `deconvblind` function supports several other optional arguments you can use to achieve the best possible result, such as specifying a damping parameter to handle additive noise in the blurred image. To see the impact of these optional arguments, as well as the functional option that allows you to place additional constraints on the PSF reconstruction, see the Image Processing Toolbox Deblurring Demos.

Creating Your Own Deblurring Functions

All the toolbox deblurring functions perform deconvolution in the frequency domain, where the process becomes a simple matrix multiplication. To work in the frequency domain, the deblurring functions must convert the PSF you provide into an optical transfer function (OTF), using the `psf2otf` function. The toolbox also provides a function to convert an OTF into a PSF, `otf2psf`. The toolbox makes these functions available in case you want to create your own deblurring functions. (In addition, to aid this conversion between PSFs and OTFs, the toolbox also makes the padding function, `padarray`, available.)

Avoiding Ringing in Deblurred Images

The Discrete Fourier Transform (DFT), used by the deblurring functions, assumes that the frequency pattern of an image is periodic. This assumption creates a high-frequency drop-off at the edges of images. In the figure, the shaded area represents the actual extent of the image; the unshaded area represents the assumed periodicity.



This high-frequency drop-off can create an effect called *boundary related ringing* in deblurred images. In this figure, note the horizontal and vertical patterns in the image.



To avoid ringing, use the `edgetaper` function to preprocess your images before passing them to the deblurring functions. The `edgetaper` function removes the high-frequency drop-off at the edge of an image by blurring the entire image and then replacing the center pixels of the blurred image with the original image. In this way, the edges of the image taper off to a lower frequency.

13

Color

This section describes the toolbox functions that help you work with color image data. Note that "color" includes shades of gray; therefore much of the discussion in this chapter applies to grayscale images as well as color images. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Approximation	The method by which the software chooses replacement colors in the event that direct matches cannot be found. The methods of approximation discussed in this chapter are colormap mapping, uniform quantization, and minimum variance quantization.
Indexed image	An image whose pixel values are direct indices into an RGB colormap. In MATLAB, an indexed image is represented by an array of class `uint8`, `uint16`, or `double`. The colormap is always an m-by-3 array of class `double`. We often use the variable name `X` to represent an indexed image in memory, and `map` to represent the colormap.
Intensity image	An image consisting of intensity (grayscale) values. In MATLAB, intensity images are represented by an array of class `uint8`, `uint16`, or `double`. While intensity images are not stored with colormaps, MATLAB uses a system colormap to display them. We often use the variable name `I` to represent an intensity image in memory. This term is synonymous with the term *grayscale*.
RGB image	An image in which each pixel is specified by three values — one each for the red, blue, and green components of the pixel's color. In MATLAB, an RGB image is represented by an m-by-n-by-3 array of class `uint8`, `uint16`, or `double`. We often use the variable name `RGB` to represent an RGB image in memory.
Screen bit depth	The number of bits per screen pixel.
Screen color resolution	The number of distinct colors that can be produced by the screen.

Working with Different Screen Bit Depths

Most computer displays use 8, 16, or 24 bits per screen pixel. The number of bits per screen pixel determines the display's *screen bit depth*. The screen bit depth determines the *screen color resolution*, which is how many distinct colors the display can produce.

Regardless of the number of colors your system can display, MATLAB can store and process images with very high bit depths: 2^{24} colors for `uint8` RGB images, 2^{48} colors for `uint16` RGB images, and 2^{159} for `double` RGB images. These images display best on systems with 24-bit color, but usually look fine on 16-bit systems as well. (For additional information about how MATLAB handles color, see the MATLAB graphics documentation.)

This section:

- Describes how to determine your system's screen bit depth
- Provides guidelines for choosing a screen bit depth

Determining Your Systems Screen Bit Depth

To determine your system's screen bit depth, enter this command at the MATLAB prompt.

```
get(0,'ScreenDepth')
ans =

    16
```

The integer MATLAB returns represents the number of bits per screen pixel:

Value	Screen Bit Depth
8	8-bit displays supports 256 colors. An 8-bit display can produce any of the colors available on a 24-bit display, but only 256 distinct colors can appear at one time. (There are 256 shades of gray available, but if all 256 shades of gray are used, they take up all of the available color slots.)
16	16-bit displays usually use 5 bits for each color component, resulting in 32 (i.e., 2^5) levels each of red, green, and blue. This supports 32,768 (i.e., 2^{15}) distinct colors (of which 32 are shades of gray). Some systems use the extra bit to increase the number of levels of green that can be displayed. In this case, the number of different colors supported by a 16-bit display is actually 64,536 (i.e. 2^{16}).
24	24-bit displays use 8 bits for each of the three color components, resulting in 256 (i.e., 2^8) levels each of red, green, and blue. This supports 16,777,216 (i.e., 2^{24}) different colors. (Of these colors, 256 are shades of gray. Shades of gray occur where R=G=B.) The 16 million possible colors supported by 24-bit display can render a life-like image.
32	32-bit displays use 24 bits to store color information and use the remaining 8 bits to store transparency data (alpha channel). For information about how MATLAB supports the alpha channel, see Transparency.

Choosing a Screen Bit Depth

Depending on your system, you may be able to choose the screen bit depth you want to use. (There may be trade-offs between screen bit depth and screen color resolution.) In general, 24-bit display mode produces the best results. If you need to use a lower screen bit depth, 16-bit is generally preferable to 8-bit. However, keep in mind that a 16-bit display has certain limitations, such as:

- An image may have finer gradations of color than a 16-bit display can represent. If a color is unavailable, MATLAB uses the closest approximation.

- There are only 32 shades of gray available. If you are working primarily with grayscale images, you may get better display results using 8-bit display mode, which provides up to 256 shades of gray.

For information about reducing the number of colors used by an image, see "Reducing the Number of Colors in an Image" on page 13-6.

Reducing the Number of Colors in an Image

This section describes how to reduce the number of colors in an indexed or RGB image. A discussion is also included about dithering, which is used by the toolbox's color-reduction functions (see below.) Dithering is used to increase the apparent number of colors in an image.

The table below summarizes the Image Processing Toolbox functions for color reduction.

Function	Purpose
`imapprox`	Reduces the number of colors used by an indexed image, enabling you specify the number of colors in the new colormap.
`rgb2ind`	Converts an RGB image to an indexed image, enabling you to specify the number of colors to store in the new colormap.

On systems with 24-bit color displays, RGB (truecolor) images can display up to 16,777,216 (i.e., 2^{24}) colors. On systems with lower screen bit depths, RGB images still displays reasonably well, because MATLAB automatically uses color approximation and dithering if needed.

Indexed images, however, may cause problems if they have a large number of colors. In general, you should limit indexed images to 256 colors for the following reasons:

- On systems with 8-bit display, indexed images with more than 256 colors will need to be dithered or mapped and, therefore, may not display well.
- On some platforms, colormaps cannot exceed 256 entries.
- If an indexed image has more than 256 colors, MATLAB cannot store the image data in a `uint8` array, but generally uses an array of class `double` instead, making the storage size of the image much larger (each pixel uses 64 bits).
- Most image file formats limit indexed images to 256 colors. If you write an indexed image with more than 256 colors (using `imwrite`) to a format that does not support more than 256 colors, you will receive an error.

Using rgb2ind

`rgb2ind` converts an RGB image to an indexed image, reducing the number of colors in the process. This function provides the following methods for approximating the colors in the original image:

- Quantization
  - Uniform quantization
  - Minimum variance quantization
- Colormap mapping

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See "Dithering" on page 13-13 for a description of dithering and how to enable or disable it.

Quantization

Reducing the number of colors in an image involves *quantization*. The function `rgb2ind` uses quantization as part of its color reduction algorithm. `rgb2ind` supports two quantization methods: *uniform quantization* and *minimum variance quantization*.

An important term in discussions of image quantization is *RGB color cube*, which is used frequently throughout this section. The RGB color cube is a three-dimensional array of all of the colors that are defined for a particular data type. Since RGB images in MATLAB can be of type `uint8`, `uint16`, or `double`, three possible color cube definitions exist. For example, if an RGB image is of class `uint8`, 256 values are defined for each color plane (red, blue, and green), and, in total, there will be 2^{24} (or 16,777,216) colors defined by the color cube. This color cube is the same for all `uint8` RGB images, regardless of which colors they actually use.

The `uint8`, `uint16`, and `double` color cubes all have the same range of colors. In other words, the brightest red in an `uint8` RGB image displays the same as the brightest red in a `double` RGB image. The difference is that the `double` RGB color cube has many more shades of red (and many more shades of all colors). Figure 13-1, below, shows an RGB color cube for a `uint8` image.

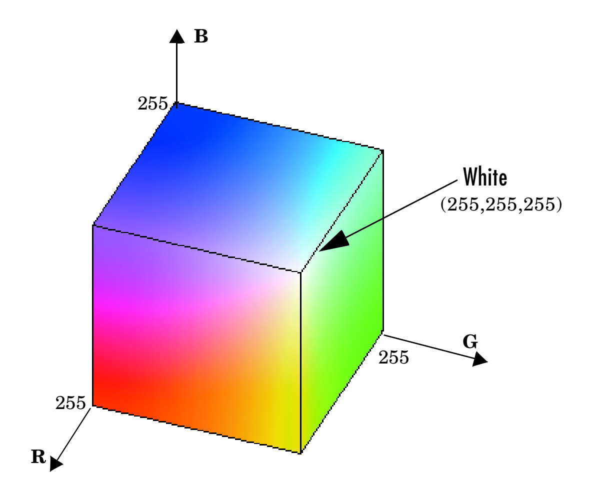


Figure 13-1: RGB Color Cube for uint8 Images

Quantization involves dividing the RGB color cube into a number of smaller boxes, and then mapping all colors that fall within each box to the color value at the *center* of that box.

Uniform quantization and minimum variance quantization differ in the approach used to divide up the RGB color cube. With uniform quantization, the color cube is cut up into equal-sized boxes (smaller cubes). With minimum variance quantization, the color cube is cut up into boxes (not necessarily cubes) of different sizes; the sizes of the boxes depend on how the colors are distributed in the image.

Uniform Quantization. To perform uniform quantization, call `rgb2ind` and specify a *tolerance*. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is [0,1]. For example, if you specify a tolerance of `0.1`, the edges of the boxes are one-tenth the length of the RGB color cube and the maximum total number of boxes is

```
n = (floor(1/tol)+1)^3
```

The commands below perform uniform quantization with a tolerance of 0.1.

```
RGB = imread('flowers.tif');
[x,map] = rgb2ind(RGB, 0.1);
```

Figure 13-2 illustrates uniform quantization of a `uint8` image. For clarity, the figure shows a two-dimensional slice (or color plane) from the color cube where Red=0, and Green and Blue range from 0 to 255. The actual pixel values are denoted by the centers of the x's.

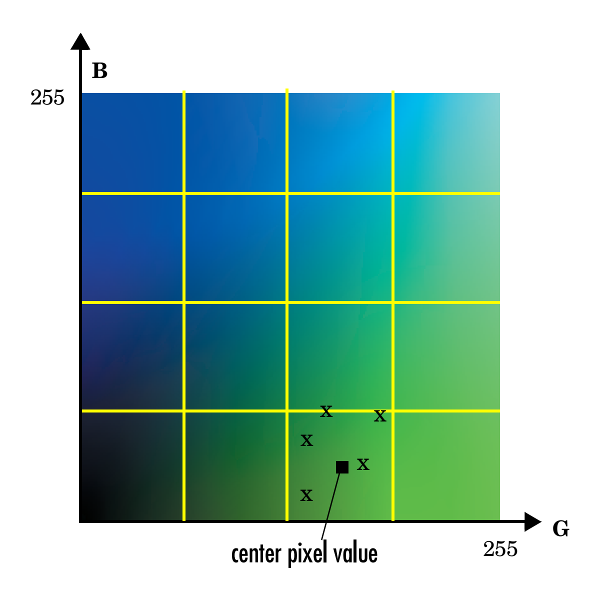


Figure 13-2: Uniform Quantization on a Slice of the RGB Color Cube

After the color cube has been divided, all empty boxes are thrown out. Therefore, only one of the boxes in Figure 13-2 is used to produce a color for the colormap. As shown earlier, the maximum length of a colormap created by uniform quantization can be predicted, but the colormap can be smaller than the prediction because `rgb2ind` removes any colors that do not appear in the input image.

Minimum Variance Quantization. To perform minimum variance quantization, call `rgb2ind` and specify the maximum number of colors in the output image's colormap. The number you specify determines the number of boxes into which

the RGB color cube is divided. These commands use minimum variance quantization to create an indexed image with 185 colors.

```
RGB = imread('flowers.tif');
[X,map] = rgb2ind(RGB,185);
```

Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixel values may be grouped together because none of their values is greater than 5 from the center pixel of the group.

In minimum variance quantization, the boxes that divide the color cube vary in size, and do not necessarily fill the color cube. If some areas of the color cube do not have pixels, there are no boxes in these areas.

While you set the number of boxes, `n`, to be used by `rgb2ind`, the placement is determined by the algorithm as it analyzes the color data in your image. Once the image is divided into `n` optimally located boxes, the pixels within each box are mapped to the pixel value at the center of the box, as in uniform quantization.

The resulting colormap usually has the number of entries you specify. This is because the color cube is divided so that each region contains at least one color that appears in the input image. If the input image uses fewer colors than the number you specify, the output colormap will have fewer than `n` colors, and the output image will contain all of the colors of the input image.

Figure 13-3 shows the same two-dimensional slice of the color cube as was used in Figure 13-2 (for demonstrating uniform quantization). Eleven boxes have been created using minimum variance quantization.

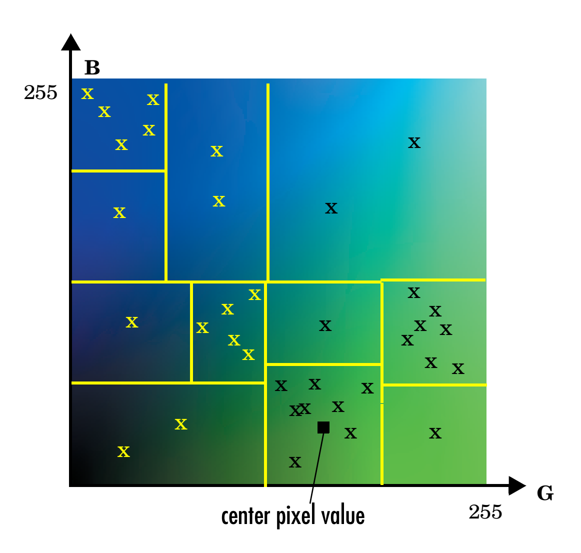


Figure 13-3: Minimum Variance Quantization on a Slice of the RGB Color Cube

For a given number of colors, minimum variance quantization produces better results than uniform quantization, because it takes into account the actual data. Minimum variance quantization allocates more of the colormap entries to colors that appear frequently in the input image. It allocates fewer entries to colors that appear infrequently. As a result, the accuracy of the colors is higher than with uniform quantization. For example, if the input image has many shades of green and few shades of red, there will be more greens than reds in the output colormap. Note that the computation for minimum variance quantization takes longer than that for uniform quantization.

Colormap Mapping

If you specify an actual colormap to use, `rgb2ind` uses *colormap mapping* (instead of quantization) to find the colors in the specified colormap that best match the colors in the RGB image. This method is useful if you need to create images that use a fixed colormap. For example, if you want to display multiple indexed images on an 8-bit display, you can avoid color problems by mapping them all to the same colormap. Colormap mapping produces a good approximation if the specified colormap has similar colors to those in the RGB

image. If the colormap does not have similar colors to those in the RGB image, this method produces poor results.

This example illustrates mapping two images to the same colormap. The colormap used for the two images is created on the fly using the MATLAB function `colorcube`, which creates an RGB colormap containing the number of colors that you specify. (`colorcube` always creates the same colormap for a given number of colors.) Because the colormap includes colors all throughout the RGB color cube, the output images can reasonably approximate the input images.

```
RGB1 = imread('autumn.tif');
RGB2 = imread('flowers.tif');
X1 = rgb2ind(RGB1,colorcube(128));
X2 = rgb2ind(RGB2,colorcube(128));
```

Note The function `subimage` is also helpful for displaying multiple indexed images. For more information see "Displaying Multiple Images in the Same Figure" on page 3-20 or the reference page for `subimage`.

Reducing Colors in an Indexed Image

Use `imapprox` when you need to reduce the number of colors in an indexed image. `imapprox` is based on `rgb2ind` and uses the same approximation methods. Essentially, `imapprox` first calls `ind2rgb` to convert the image to RGB format, and then calls `rgb2ind` to return a new indexed image with fewer colors.

For example, these commands create a version of the `trees` image with 64 colors, rather than the original 128.

```
load trees
[Y,newmap] = imapprox(X,map,64);
imshow(Y, newmap);
```

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See "Dithering" on page 13-13 for a description of dithering and how to enable or disable it.

Dithering

When you use rgb2ind or imapprox to reduce the number of colors in an image, the resulting image may look inferior to the original, because some of the colors are lost. rgb2ind and imapprox both perform *dithering* to increase the apparent number of colors in the output image. Dithering changes the colors of pixels in a neighborhood so that the average color in each neighborhood approximates the original RGB color.

For an example of how dithering works, consider an image that contains a number of dark pink pixels for which there is no exact match in the colormap. To create the appearance of this shade of pink, the Image Processing Toolbox selects a combination of colors from the colormap, that, taken together as a six-pixel group, approximate the desired shade of pink. From a distance, the pixels appear to be correct shade, but if you look up close at the image, you can see a blend of other shades, perhaps red and pale pink pixels. The commands below load a 24-bit image, and then use rgb2ind to create two indexed images with just eight colors each.

```
rgb=imread('lily.tif');
imshow(rgb);
[X_no_dither,map]=rgb2ind(rgb,8,'nodither');
[X_dither,map]=rgb2ind(rgb,8,'dither');
figure, imshow(X_no_dither,map);
figure, imshow(X_dither,map);
```



Figure 13-4: Examples of Color Reduction with and Without Dithering

Notice that the dithered image has a larger number of apparent colors but is somewhat fuzzy-looking. The image produced without dithering has fewer apparent colors, but an improved spatial resolution when compared to the dithered image. One risk in doing color reduction without dithering is that the new image my contain false contours (see the rose in the upper-right corner).

Converting to Other Color Spaces

The Image Processing Toolbox represents colors as RGB values, either directly (in an RGB image) or indirectly (in an indexed image, where the colormap is stored in RGB format). However, there are other models besides RGB for representing colors numerically. For example, a color can be represented by its hue, saturation, and value components (HSV) instead. The various models for color data are called *color spaces.*

The functions in the Image Processing Toolbox that work with color assume that images use the RGB color space. However, the toolbox provides support for other color spaces though a set of conversion functions. You can use these functions to convert between RGB and the following color spaces:

- National Television Systems Committee (NTSC)
- YCbCr
- Hue, saturation, value (HSV)

These section describes these color spaces and the conversion routines for working with them:

- "NTSC Color Space"
- "YCbCr Color Space" on page 13-16
- "HSV Color Space" on page 13-16

NTSC Color Space

The NTSC color space is used in televisions in the United States. One of the main advantages of this format is that grayscale information is separated from color data, so the same signal can be used for both color and black and white sets. In the NTSC format, image data consists of three components: luminance (Y), hue (I), and saturation (Q). The first component, luminance, represents grayscale information, while the last two components make up chrominance (color information).

The function `rgb2ntsc` converts colormaps or RGB images to the NTSC color space. `ntsc2rgb` performs the reverse operation.

For example, these commands convert the `flowers` image to NTSC format.

```
RGB = imread('flowers.tif');
YIQ = rgb2ntsc(RGB);
```

Because luminance is one of the components of the NTSC format, the RGB to NTSC conversion is also useful for isolating the gray level information in an image. In fact, the toolbox functions `rgb2gray` and `ind2gray` use the `rgb2ntsc` function to extract the grayscale information from a color image.

For example, these commands are equivalent to calling `rgb2gray`.

```
YIQ = rgb2ntsc(RGB);
I = YIQ(:,:,1);
```

Note In YIQ color space, I is one of the two color components, not the grayscale component.

YCbCr Color Space

The YCbCr color space is widely used for digital video. In this format, luminance information is stored as a single component (Y), and chrominance information is stored as two color-difference components (Cb and Cr). Cb represents the difference between the blue component and a reference value. Cr represents the difference between the red component and a reference value.

YCbCr data can be double precision, but the color space is particularly well suited to `uint8` data. For `uint8` images, the data range for Y is [16, 235], and the range for Cb and Cr is [16, 240]. YCbCr leaves room at the top and bottom of the full `uint8` range so that additional (nonimage) information can be included in a video stream.

The function `rgb2ycbcr` converts colormaps or RGB images to the YCbCr color space. `ycbcr2rgb` performs the reverse operation.

For example, these commands convert the `flowers` image to YCbCr format.

```
RGB = imread('flowers.tif');
YCBCR = rgb2ycbcr(RGB);
```

HSV Color Space

The HSV color space (hue, saturation, value) is often used by people who are selecting colors (e.g., of paints or inks) from a color wheel or palette, because it corresponds better to how people experience color than the RGB color space

does. The functions `rgb2hsv` and `hsv2rgb` convert images between the RGB and HSV color spaces.

As hue varies from 0 to 1.0, the corresponding colors vary from red, through yellow, green, cyan, blue, and magenta, back to red, so that there are actually red values both at 0 and 1.0. As saturation varies from 0 to 1.0, the corresponding colors (hues) vary from unsaturated (shades of gray) to fully saturated (no white component). As value, or brightness, varies from 0 to 1.0, the corresponding colors become increasingly brighter.

Figure 13-5 illustrates the HSV color space.

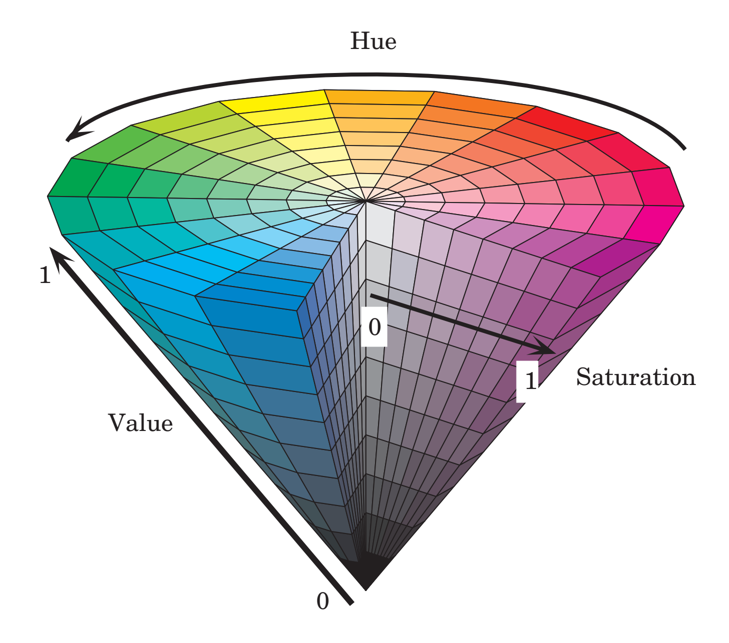


Figure 13-5: Illustration of the HSV Color Space

The function `rgb2hsv` converts colormaps or RGB images to the HSV color space. `hsv2rgb` performs the reverse operation. These commands convert an RGB image to HSV color space.

```
RGB = imread('flowers.tif');
HSV = rgb2hsv(RGB);
```

For closer inspection of the HSV color space, the next block of code displays the separate color planes (hue, saturation, and value) of an HSV image.

```
RGB=reshape(ones(64,1)*reshape(jet(64),1,192),[64,64,3]);
HSV=rgb2hsv(RGB);
H=HSV(:,:,1);
S=HSV(:,:,2);
V=HSV(:,:,3);
imshow(H)
figure, imshow(S);
figure, imshow(V);
figure, imshow(RGB);
```

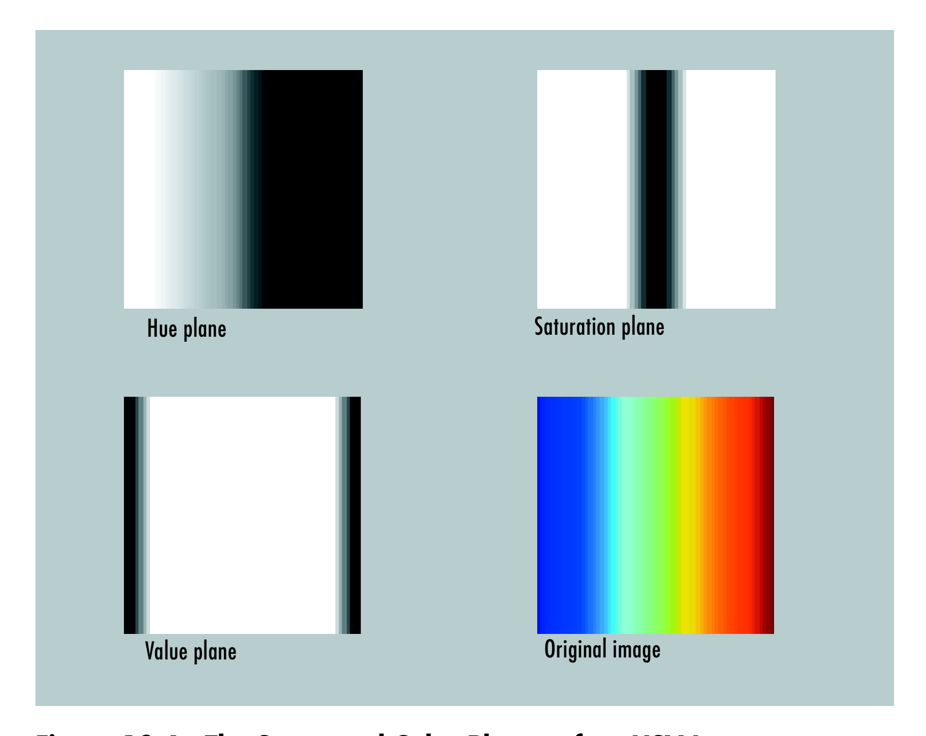


Figure 13-6: The Separated Color Planes of an HSV Image

The images in Figure 13-6 can be scrutinized for a better understanding of how the HSV color space works. As you can see by looking at the hue plane image, hue values make a nice linear transition from high to low. If you compare the hue plane image against the original image, you can see that shades of deep blue have the highest values, and shades of deep red have the lowest values. (In actuality, there are values of red on both ends of the hue scale, which you can see if you look back at the model of the HSV color space in Figure 13-5. To avoid confusion, our sample image uses only the red values from the *beginning* of the hue range.) Saturation can be thought of as the purity of a color. As the saturation plane image shows, the colors with the highest saturation have the highest values and are represented as white. In the center of the saturation image, notice the various shades of gray. These correspond to a mixture of colors; the cyans, greens, and yellow shades are mixtures of true colors. Value is roughly equivalent to brightness, and you will notice that the brightest areas of the value plane correspond to the brightest colors in the original image.

14

Function Reference

This section describes the Image Processing Toolbox functions.

Functions – By Category

This section provides brief descriptions of all the functions in the Image Processing Toolbox. The functions are listed in tables in the following broad categories

If you know the name of a function, go directly to the "Functions – Alphabetical List" section to view its reference page.

"Image Input, Output, and Display"	Functions for importing, exporting, and displaying images and converting between image formats
"Spatial Transformation and Registration"	Functions for performing spatial transformations and image registration
"Image Analysis and Statistics"	Functions for performing image analysis and getting pixel values and statistics
"Image Enhancement and Restoration"	Functions for image enhancement and restoration, such as deblurring.
"Linear Filtering and Transforms"	Functions for creating and using linear filters and transform
"Morphological Operations"	Functions for performing morphological image processing
"Region-Based, Neighborhood, and Block Processing"	Functions to define regions of interest and operate on these regions
"Colormap and Color Space Functions"	Functions for working with image color
"Miscellaneous Functions"	functions that perform image arithmetic, array operations, and set and get Image Processing Toolbox preferences

Image Input, Output, and Display

- "Image Display"
- "Image File I/O"

- "Image Types and Type Conversions"

Image Display

`colorbar`	Display colorbar. (This is a MATLAB function.)
`getimage`	Get image data from axes
`image`	Create and display image object. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`imagesc`	Scale data and display as image. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`immovie`	Make movie from multiframe indexed image
`imshow`	Display image
`montage`	Display multiple image frames as rectangular montage
`subimage`	Display multiple images in single figure
`truesize`	Adjust display size of image
`warp`	Display image as texture-mapped surface
`zoom`	Zoom in and out of image or 2-D plot. (This is a MATLAB function.)

Image File I/O

`dicominfo`	Read metadata from a DICOM message
`dicomread`	Read a DICOM image
`imfinfo`	Return information about image file. (This is a MATLAB function.)
`imread`	Read image file. (This is a MATLAB function.)
`imwrite`	Write image file. (This is a MATLAB function.)

Image Types and Type Conversions

`dither`	Convert image using dithering
`double`	Convert data to double precision. (This is a MATLAB function.)
`gray2ind`	Convert intensity image to indexed image
`grayslice`	Create indexed image from intensity image by thresholding
`graythresh`	Compute global image threshold using Otsu's method
`im2bw`	Convert image to binary image by thresholding
`im2double`	Convert image array to double precision
`im2java`	Convert image to instance of Java image object
`im2uint16`	Convert image array to 16-bit unsigned integers
`im2uint8`	Convert image array to 8-bit unsigned integers
`ind2gray`	Convert indexed image to intensity image
`ind2rgb`	Convert indexed image to RGB image
`isbw`	Return true for binary image
`isgray`	Return true for intensity image
`isind`	Return true for indexed image
`isrgb`	Return true for RGB image
`label2rgb`	Convert a label matrix to an RGB image

`mat2gray`	Convert matrix to intensity image
`rgb2gray`	Convert RGB image or colormap to grayscale
`rgb2ind`	Convert RGB image to indexed image
`uint16`	Convert data to unsigned 16-bit integers. (This is a MATLAB function.)
`uint8`	Convert data to unsigned 8-bit integers. (This is a MATLAB function.)

Spatial Transformation and Registration

- "Spatial Transformations"
- "Image Registration"

Spatial Transformations

`checkerboard`	Create checkerboard image
`findbounds`	Find output bounds for spatial transformation
`fliptform`	Flip the input and output roles of a TFORM structure
`imcrop`	Crop image
`imresize`	Resize image
`imrotate`	Rotate image
`interp2`	2-D data interpolation. (This is a MATLAB function. See the online MATLAB Function Reference for its reference page.)
`imtransform`	Apply 2-D spatial transformation to image
`makeresampler`	Create resampling structure
`maketform`	Create geometric transformation structure
`tformarray`	Geometric transformation of a multi-dimensional array
`tformfwd`	Apply forward geometric transformation
`tforminv`	Apply inverse geometric transformation

Image Registration

`cpcorr`	Tune control point locations using cross-correlation
`cp2tform`	Infer geometric transformation from control point pairs
`cpselect`	Control point selection tool
`cpstruct2pairs`	Convert `CPSTRUCT` to valid pairs of control points
`normxcorr2`	Normalized two-dimensional cross-correlation

Image Analysis and Statistics

- "Image Analysis"
- "Pixel Values and Statistics"

Image Analysis

`edge`	Find edges in intensity image
`qtdecomp`	Perform quadtree decomposition
`qtgetblk`	Get block values in quadtree decomposition
`qtsetblk`	Set block values in quadtree decomposition

Pixel Values and Statistics

`corr2`	Compute 2-D correlation coefficient
`imcontour`	Create contour plot of image data
`imfeature`	Compute feature measurements for image regions
`imhist`	Display histogram of image data
`impixel`	Determine pixel color values
`improfile`	Compute pixel-value cross-sections along line segments
`mean2`	Compute mean of matrix elements
`pixval`	Display information about image pixels
`regionprops`	Measure properties of image regions
`std2`	Compute standard deviation of matrix elements

Image Enhancement and Restoration

• "Image Enhancement"
• "Image Restoration (Deblurring)"

Image Enhancement

histeq	Enhance contrast using histogram equalization
imadjust	Adjust image intensity values or colormap
imnoise	Add noise to an image
medfilt2	Perform 2-D median filtering
ordfilt2	Perform 2-D order-statistic filtering
stretchlim	Find limits to contrast stretch an image
wiener2	Perform 2-D adaptive noise-removal filtering

Image Restoration (Deblurring)

deconvblind	Restore image using blind deconvolution
deconvlucy	Restore image using accelerated Richardson-Lucy algorithm
deconvreg	Restore image using Regularized filter
deconvwnr	Restore image using Wiener filter
edgetaper	Taper the discontinuities along the image edges
otf2psf	Convert optical transfer function to point-spread function
psf2otf	Convert point-spread function to optical transfer function

Linear Filtering and Transforms

- "Linear Filtering"
- "Linear 2-D Filter Design"
- "Image Transforms"

Linear Filtering

`conv2`	Perform 2-D convolution. (This is a MATLAB function.)
`convmtx2`	Compute 2-D convolution matrix
`convn`	Perform N-D convolution. (This is a MATLAB function.)
`filter2`	Perform 2-D filtering. (This is a MATLAB function.)
`fspecial`	Create predefined filters
`imfilter`	Multidimensional image filtering

Linear 2-D Filter Design

`freqspace`	Determine 2-D frequency response spacing. (This is a MATLAB function.)
`freqz2`	Compute 2-D frequency response
`fsamp2`	Design 2-D FIR filter using frequency sampling
`ftrans2`	Design 2-D FIR filter using frequency transformation
`fwind1`	Design 2-D FIR filter using 1-D window method
`fwind2`	Design 2-D FIR filter using 2-D window method

Image Transforms

`dct2`	Compute 2-D discrete cosine transform
`dctmtx`	Compute discrete cosine transform matrix
`fft2`	Compute 2-D fast Fourier transform. (This is a MATLAB function.)
`fftn`	Compute N-D fast Fourier transform. (This is a MATLAB function.)
`fftshift`	Reverse quadrants of output of FFT. (This is a MATLAB function.)
`idct2`	Compute 2-D inverse discrete cosine transform
`ifft2`	Compute 2-D inverse fast Fourier transform. (This is a MATLAB function.)

ifftn	Compute N-D inverse fast Fourier transform. (This is a MATLAB function.)
iradon	Compute inverse Radon transform
phantom	Generate a head phantom image
radon	Compute Radon transform

Morphological Operations

- "Intensity and Binary Images"
- "Binary Images"
- "Structuring Element (STREL) Creation and Manipulation"

Intensity and Binary Images

`conndef`	Default connectivity array
`imbothat`	Perform bottom-hat filtering
`imclearborder`	Suppress light structures connected to image border
`imclose`	Close image
`imdilate`	Dilate image
`imerode`	Erode image
`imextendedmax`	Extended-maxima transform
`imextendedmin`	Extended-minima transform
`imfill`	Fill image regions
`imhmax`	H-maxima transform
`imhmin`	H-minima transform
`imimposemin`	Impose minima
`imopen`	Open image
`imreconstruct`	Perform morphological reconstruction
`imregionalmax`	Regional maxima of image
`imregionalmin`	Regional minima of image
`imtophat`	Perform tophat filtering
`watershed`	Find image watershed regions

Binary Images

`applylut`	Perform neighborhood operations using lookup tables
`bwarea`	Area of objects in binary image
`bwareaopen`	Binary area open; remove small objects
`bwdist`	Distance transform
`bweuler`	Euler number of binary image
`bwfill`	Fill background regions in binary image

`bwhitmiss`	Binary hit-miss operation
`bwlabel`	Label connected components in 2-D binary image
`bwlabeln`	Label connected components in N-D binary image.
`bwmorph`	Perform morphological operations on binary image
`bwpack`	Pack binary image
`bwperim`	Find perimeter of objects in binary image
`bwselect`	Select objects in binary image
`bwulterode`	Ultimate erosion
`bwunpack`	Unpack a packed binary image
`imregionalmin`	Regional minima of image
`imtophat`	Perform tophat filtering
`makelut`	Construct lookup table for use with `applylut`

Structuring Element (STREL) Creation and Manipulation

`getheight`	Get the height of a structuring element
`getneighbors`	Get structuring element neighbor locations and heights
`getnhood`	Get structuring element neighborhood
`getsequence`	Extract sequence of decomposed structuring elements
`isflat`	Return true for flat structuring element
`reflect`	Reflect structuring element
`strel`	Create morphological structuring element
`translate`	Translate structuring element

Region-Based, Neighborhood, and Block Processing

- "Region-Based Processing"
- "Neighborhood and Block Processing"

Region-Based Processing

`roicolor`	Select region of interest, based on color
`roifill`	Smoothly interpolate within arbitrary region
`roifilt2`	Filter a region of interest
`roipoly`	Select polygonal region of interest

Neighborhood and Block Processing

`bestblk`	Choose block size for block processing
`blkproc`	Implement distinct block processing for image
`col2im`	Rearrange matrix columns into blocks
`colfilt`	Perform neighborhood operations using columnwise functions
`im2col`	Rearrange image blocks into columns
`nlfilter`	Perform general sliding-neighborhood operations

Colormap and Color Space Functions

- “Colormap Manipulation”
- “Color Space Conversions”

Colormap Manipulation

`brighten`	Brighten or darken colormap. (This is a MATLAB function.)
`cmpermute`	Rearrange colors in colormap
`cmunique`	Find unique colormap colors and corresponding image
`colormap`	Set or get color lookup table. (This is a MATLAB function.)
`imapprox`	Approximate indexed image by one with fewer colors
`rgbplot`	Plot RGB colormap components. (This is a MATLAB function.)

Color Space Conversions

`hsv2rgb`	Convert HSV values to RGB color space. (This is a MATLAB function.)
`ntsc2rgb`	Convert NTSC values to RGB color space
`rgb2hsv`	Convert RGB values to HSV color space. (This is a MATLAB function.)
`rgb2ntsc`	Convert RGB values to NTSC color space
`rgb2ycbcr`	Convert RGB values to YCbCr color space
`ycbcr2rgb`	Convert YCbCr values to RGB color space

Miscellaneous Functions

- “Image Arithmetic”
- “Toolbox Preferences”
- “Array Operations”

Image Arithmetic

`imabsdiff`	Compute absolute difference of two images
`imadd`	Add two images, or add constant to image
`imcomplement`	Complement image
`imdivide`	Divide two images, or divide image by constant.
`imlincomb`	Compute linear combination of images
`immultiply`	Multiply two images, or multiply image by constant
`imsubtract`	Subtract two images, or subtract constant from image

Toolbox Preferences

`iptgetpref`	Get value of Image Processing Toolbox preference
`iptsetpref`	Set value of Image Processing Toolbox preference

Array Operations

`padarray`	Pad an array

Functions – Alphabetical List

This section contains detailed descriptions of all toolbox functions. Each function reference page contains some or all of this information:

- The function name
- The purpose of the function
- The function syntax

  All valid input argument and output argument combinations are shown. In some cases, an ellipsis (. . .) is used for the input arguments. This means that all preceding input argument combinations are valid for the specified output argument(s).
- A description of each argument
- A description of the function
- Additional remarks about usage
- An example of usage
- Related functions

Purpose Perform neighborhood operations on binary images using lookup tables

Syntax

```
A = applylut(BW,LUT)
```

Description `A = applylut(BW,LUT)` performs a 2-by-2 or 3-by-3 neighborhood operation on binary image `BW` by using a lookup table (`LUT`). `LUT` is either a 16-element or 512-element vector returned by `makelut`. The vector consists of the output values for all possible 2-by-2 or 3-by-3 neighborhoods.

Class Support `BW` can be numeric or logical, and it must be real, two-dimensional, and nonsparse. `LUT` can be numeric or logical, and it must be a real vector with 16 or 512 elements. If all the elements of `LUT` are `0` or `1`, then `A` is logical. If all the elements of `LUT` are integers between 0 and 255, then `A` is `uint8`. For all other cases, `A` is `double`.

Algorithm `applylut` performs a neighborhood operation on a binary image by producing a matrix of indices into `lut`, and then replacing the indices with the actual values in `lut`. The specific algorithm used depends on whether you use 2-by-2 or 3-by-3 neighborhoods.

2-by-2 Neighborhoods

For 2-by-2 neighborhoods, `length(lut)` is 16. There are four pixels in each neighborhood, and two possible states for each pixel, so the total number of permutations is $2^4 = 16$.

To produce the matrix of indices, `applylut` convolves the binary image `BW` with this matrix.

```
8     2
4     1
```

The resulting convolution contains integer values in the range [0,15]. `applylut` uses the central part of the convolution, of the same size as `BW`, and adds 1 to each value to shift the range to [1,16]. It then constructs `A` by replacing the values in the cells of the index matrix with the values in `lut` that the indices point to.

3-by-3 Neighborhoods

For 3-by-3 neighborhoods, `length(lut)` is 512. There are nine pixels in each neighborhood, and 2 possible states for each pixel, so the total number of permutations is $2^9 = 512$.

To produce the matrix of indices, `applylut` convolves the binary image `BW` with this matrix.

```
256    32    4
128    16    2
 64     8    1
```

The resulting convolution contains integer values in the range [0,511]. `applylut` uses the central part of the convolution, of the same size as `BW`, and adds 1 to each value to shift the range to [1,512]. It then constructs `A` by replacing the values in the cells of the index matrix with the values in `lut` that the indices point to.

Example

In this example, you perform erosion using a 2-by-2 neighborhood. An output pixel is `on` only if all four of the input pixel's neighborhood pixels are `on`.

```
lut = makelut('sum(x(:)) == 4',2);
BW1 = imread('text.tif');
BW2 = applylut(BW1,lut);
imshow(BW1)
figure, imshow(BW2)
```



See Also

`makelut`

Purpose

Determine block size for block processing

Syntax

```
siz = bestblk([m n],k)
[mb,nb] = bestblk([m n],k)
```

Description

`siz = bestblk([m n],k)` returns, for an `m`-by-`n` image, the optimal block size for block processing. `k` is a scalar specifying the maximum row and column dimensions for the block; if the argument is omitted, it defaults to 100. The return value, `siz`, is a 1-by-2 vector containing the row and column dimensions for the block.

`[mb,nb] = bestblk([m n],k)` returns the row and column dimensions for the block in `mb` and `nb`, respectively.

Algorithm

`bestblk` returns the optimal block size given `m`, `n`, and `k`. The algorithm for determining `siz` is:

- If `m` is less than or equal to `k`, return `m`.
- If `m` is greater than `k`, consider all values between `min(m/10,k/2)` and `k`. Return the value that minimizes the padding required.

The same algorithm is then repeated for `n`.

Example

```
siz = bestblk([640 800],72)

siz =

    64    50
```

See Also

`blkproc`

blkproc

Purpose Implement distinct block processing for an image

Syntax

```
B = blkproc(A,[m n],fun)
B = blkproc(A,[m n],fun,P1,P2,...)
B = blkproc(A,[m n],[mborder nborder],fun,...)
B = blkproc(A,'indexed',...)
```

Description

`B = blkproc(A,[m n],fun)` processes the image `A` by applying the function `fun` to each distinct `m`-by-`n` block of `A`, padding `A` with zeros if necessary. `fun` is a function that accepts an `m`-by-`n` matrix, `x`, and returns a matrix, vector, or scalar `y`.

```
y = fun(x)
```

`blkproc` does not require that `y` be the same size as `x`. However, `B` is the same size as `A` only if `y` is the same size as `x`.

`B = blkproc(A,[m n],fun,P1,P2,...)` passes the additional parameters `P1,P2,...,` to `fun`.

`B = blkproc(A,[m n],[mborder nborder],fun,...)` defines an overlapping border around the blocks. `blkproc` extends the original `m`-by-`n` blocks by `mborder` on the top and bottom, and `nborder` on the left and right, resulting in blocks of size `(m+2*mborder)`-by-`(n+2*nborder)`. The `blkproc` function pads the border with zeros, if necessary, on the edges of `A`. The function `fun` should operate on the extended block.

The line below processes an image matrix as 4-by-6 blocks, each having a row border of 2 and a column border of 3. Because each 4-by-6 block has this 2-by-3 border, `fun` actually operates on blocks of size 8-by-12.

```
B = blkproc(A,[4 6],[2 3],fun,...)
```

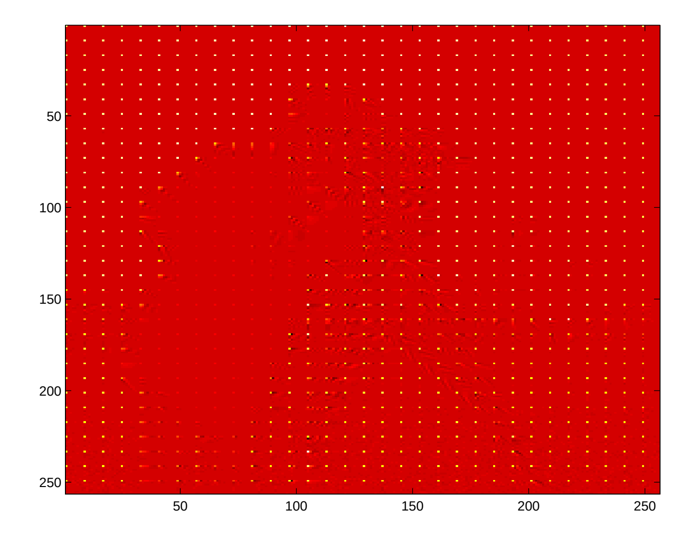
`B = blkproc(A,'indexed',...)` processes `A` as an indexed image, padding with zeros if the class of `A` is `uint8` or `uint16`, or ones if the class of `A` is `double`.

Class Support The input image `A` can be of any class supported by `fun`. The class of `B` depends on the class of the output from `fun`.

Example

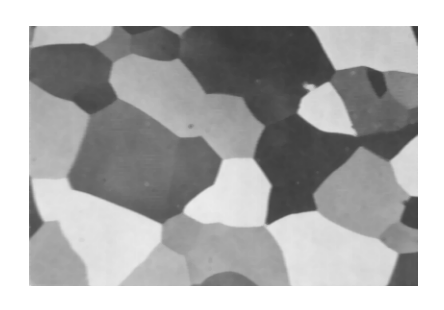
fun can be a function_handle created using @. This example uses blkproc to compute the 2-D DCT of each 8-by-8 block to the standard deviation of the elements in that block.

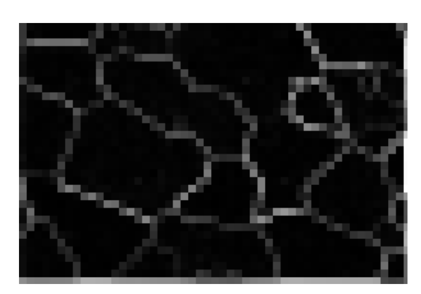
```
I = imread('cameraman.tif');
fun = @dct2;
J = blkproc(I,[8 8],fun);
imagesc(J), colormap(hot)
```



fun can also be an inline object. This example uses blkproc to set the pixels in each 8-by-8 block to the standard deviation of the elements in that block.

```
I = imread('alumgrns.tif');
fun = inline('std2(s)*ones(size(x))');
I2 = blkproc(I,[8 8],'std2(x)*ones(size(x))');
imshow(I)
figure, imshow(I2,[]);
```





See Also `colfilt`, `nlfilter`, `inline`

Purpose

Brighten or darken a colormap

`brighten` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

bwarea

Purpose

Compute the area of the objects in a binary image

Syntax

```
total = bwarea(BW)
```

Description

`total = bwarea(BW)` estimates the area of the objects in binary image `BW`. `total` is a scalar whose value corresponds roughly to the total number of `on` pixels in the image, but may not be exactly the same because different patterns of pixels are weighted differently.

Class Support

`BW` can be numeric or logical. For numeric input, any non-zero pixels are considered to be "on". The return value, `total`, is of class `double`.

Algorithm

`bwarea` estimates the area of all of the `on` pixels in an image by summing the areas of each pixel in the image. The area of an individual pixel is determined by looking at its 2-by-2 neighborhood. There are six different patterns, each representing a different area:

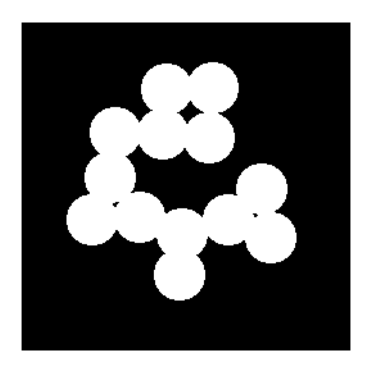
- Patterns with zero `on` pixels (area = 0)
- Patterns with one `on` pixel (area = 1/4)
- Patterns with two adjacent `on` pixels (area = 1/2)
- Patterns with two diagonal `on` pixels (area = 3/4)
- Patterns with three `on` pixels (area = 7/8)
- Patterns with all four `on` pixels (area = 1)

Keep in mind that each pixel is part of four different 2-by-2 neighborhoods. This means, for example, that a single `on` pixel surrounded by `off` pixels has a total area of 1.

Example

This example computes the area in the objects of a 256-by-256 binary image.

```
BW = imread('circles.tif');
imshow(BW);
```



```
bwarea(BW)

ans =

      15799
```

See Also `bweuler`, `bwperim`

References Pratt, William K. *Digital Image Processing*. New York: John Wiley & Sons, Inc., 1991. p. 634.

bwareaopen

Purpose

Binary area open; remove small objects

Syntax

```
BW2 = bwareaopen(BW,P)
BW2 = bwareaopen(BW,P,CONN)
```

Description

`BW2 = bwareaopen(BW,P)` removes from a binary image all connected components (objects) that have fewer than `P` pixels, producing another binary image `BW2`. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW),'maximal')` for higher dimensions.

`BW2 = bwareaopen(BW,P,CONN)` specifies the desired connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by-...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

`BW` can be a logical or numeric array of any dimension, and it must be nonsparse. The return value, `BW2`, is of class `logical`.

Algorithm

The basic steps are:

1. Determine the connected components.

   ```
   L = bwlabeln(BW, CONN);
   ```

2 Compute the area of each component.

```
S = regionprops(L, 'Area');
```

3 Remove small objects.

```
bw2 = ismember(L, find([S.Area] >= P));
```

Example

Remove all objects containing fewer than 40 pixels in an image.

1 Read in the image and display it.

```
bw = imread('text.tif');
imshow(bw)
```

2 Remove all objects smaller than 40 pixels. Note the missing letters.

```
bw2 = bwareaopen(bw,40);
figure, imshow(bw2)
```

See Also bwlabel, bwlabeln, conndef, regionprops.

Purpose

Distance transform

Syntax

```
D = bwdist(BW)
[D,L] = bwdist(BW)
[D,L] = bwdist(BW,METHOD)
```

Description

D = bwdist(BW) computes the Euclidean distance transform of the binary image BW. For each pixel in BW, the distance transform assigns a number that is the distance between that pixel and the nearest nonzero pixel of BW. bwdist uses the Euclidean distance metric by default. BW can have any dimension. D is the same size as BW.

[D,L] = bwdist(BW) also computes the nearest-neighbor transform and returns it as a label matrix, L, which has the same size as BW and D. Each element of L contains the linear index of the nearest nonzero pixel of BW.

[D,L] = bwdist(BW,METHOD) computes the distance transform, where METHOD specifies an alternate distance metric. METHOD can take any of these values.

'chessboard'	In 2-D, the chessboard distance between (x_1,y_1) and (x_2,y_2) is: $max(\lvert x_1-x_2\rvert, \lvert y_1-y_2\rvert)$
'cityblock'	In 2-D, the cityblock distance between (x_1,y_1) and (x_2,y_2) is: $\lvert x_1-x_2\rvert + \lvert y_1-y_2\rvert$

'euclidean'	In 2-D, the Euclidean distance between (x_1,y_1) and (x_2,y_2) is: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ This is the default method.
'quasi-euclidean'	In 2-D, the quasi-Euclidean distance between (x_1,y_1) and (x_2,y_2) is: $\|x_1 - x_2\| + (\sqrt{2} - 1)\|y_1 - y_2\|, \|x_1 - x_2\| > \|y_1 - y_2\|$ $(\sqrt{2} - 1)\|x_1 - x_2\| + \|y_1 - y_2\|, otherwise$

The METHOD string may be abbreviated.

Note bwdist uses fast algorithms to compute the true Euclidean distance transform, especially in the 2-D case. The other methods are provided primarily for pedagogical reasons. However, the alternative distance transforms are sometimes significantly faster for multidimensional input images, particularly those that have many nonzero elements.

Class support

BW can be numeric or logical, and it must be nonsparse. D and L are double matrices with the same size as BW.

Example

Here is a simple example of the Euclidean distance transform.

```
bw = zeros(5,5); bw(2,2) = 1; bw(4,4) = 1
bw =
     0     0     0     0     0
     0     1     0     0     0
     0     0     0     0     0
     0     0     0     1     0
     0     0     0     0     0

[D,L] = bwdist(bw)
```

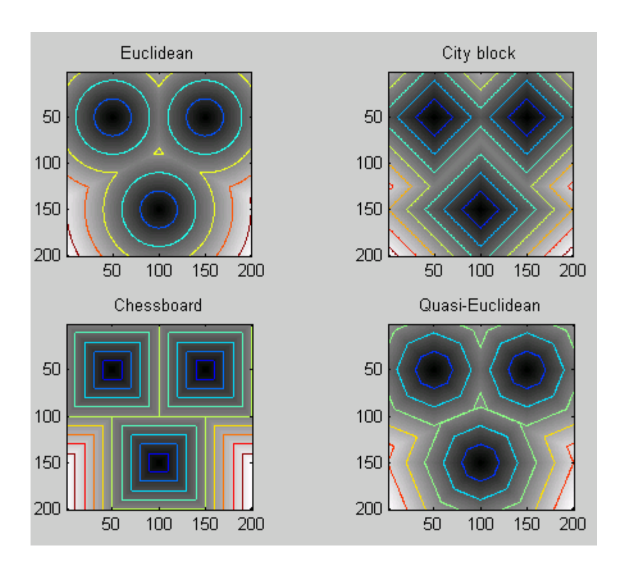
```
D =
    1.4142    1.0000    1.4142    2.2361    3.1623
    1.0000         0    1.0000    2.0000    2.2361
    1.4142    1.0000    1.4142    1.0000    1.4142
    2.2361    2.0000    1.0000         0    1.0000
    3.1623    2.2361    1.4142    1.0000    1.4142

L =
     7     7     7     7     7
     7     7     7     7    19
     7     7     7    19    19
     7     7    19    19    19
     7    19    19    19    19
```

In the nearest neighbor matrix, `L`, the values `7` and `19` represent the position of the nonzero elements using linear matrix indexing. If a pixel contains a 7, its closest nonzero neighbor is at linear position 7.

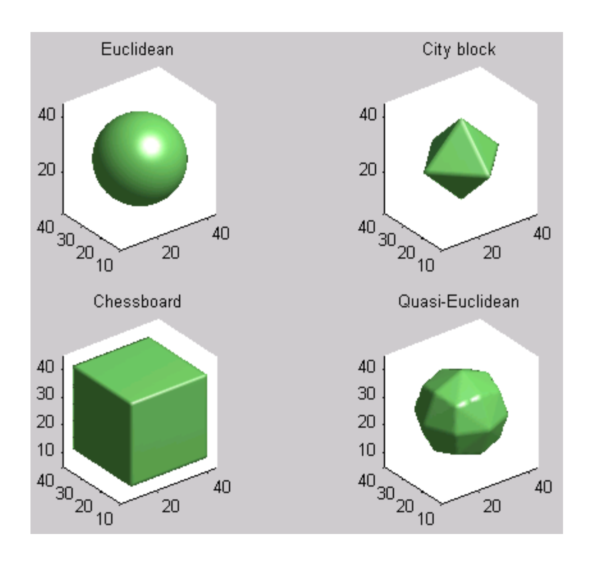
This example compares the 2-D distance transforms for the each of the supported distance methods. In the figure, note how the quasi-euclidean distance transform best approximates the circular shape achieved by the euclidean distance method.

```
bw = zeros(200,200); bw(50,50) = 1; bw(50,150) = 1;
bw(150,100) = 1;
D1 = bwdist(bw,'euclidean');
D2 = bwdist(bw,'cityblock');
D3 = bwdist(bw,'chessboard');
D4 = bwdist(bw,'quasi-euclidean');
figure
subplot(2,2,1), subimage(mat2gray(D1)), title('Euclidean')
hold on, imcontour(D1)
subplot(2,2,2), subimage(mat2gray(D2)), title('City block')
hold on, imcontour(D2)
subplot(2,2,3), subimage(mat2gray(D3)), title('Chessboard')
hold on, imcontour(D3)
subplot(2,2,4), subimage(mat2gray(D4)), title('Quasi-Euclidean')
hold on, imcontour(D4)
```



This example compares isosurface plots for the distance transforms of a 3-D image containing a single nonzero pixel in the center.

```
bw = zeros(50,50,50); bw(25,25,25) = 1;
D1 = bwdist(bw);
D2 = bwdist(bw,'cityblock');
D3 = bwdist(bw,'chessboard');
D4 = bwdist(bw,'quasi-euclidean');
figure
subplot(2,2,1), isosurface(D1,15), axis equal, view(3)
camlight, lighting gouraud, title('Euclidean')
subplot(2,2,2), isosurface(D2,15), axis equal, view(3)
camlight, lighting gouraud, title('City block')
subplot(2,2,3), isosurface(D3,15), axis equal, view(3)
camlight, lighting gouraud, title('Chessboard')
subplot(2,2,4), isosurface(D4,15), axis equal, view(3)
camlight, lighting gouraud, title('Quasi-Euclidean')
```



Algorithm

For two-dimensional Euclidean distance transforms, `bwdist` uses the second algorithm described in:

Heinz Breu, Joseph Gil, David Kirkpatrick, and Michael Werman, "Linear Time Euclidean Distance Transform Algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 5 May 1995, pp. 529-533.

For higher dimensional Euclidean distance transforms, `bwdist` uses a nearest neighbor search on an optimized kd-tree, as described in:

Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematics Software*, vol. 3, no. 3, September 1997, pp. 209-226.

For cityblock, chessboard, and quasi-Euclidean distance transforms, `bwdist` uses the two-pass, sequential scanning algorithm described in:

A. Rosenfeld and J. Pfaltz, "Sequential operations in digital picture processing," *Journal of the Association for Computing Machinery,* vol. 13, no. 4, 1966, pp. 471-494.

The different distance measures are achieved by using different sets of weights in the scans, as described in:

David Paglieroni, "Distance Transforms: Properties and Machine Vision Applications," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, vol. 54, no. 1, January 1992, pp. 57-58.

See Also `watershed`

Purpose Compute the Euler number of a binary image

Syntax

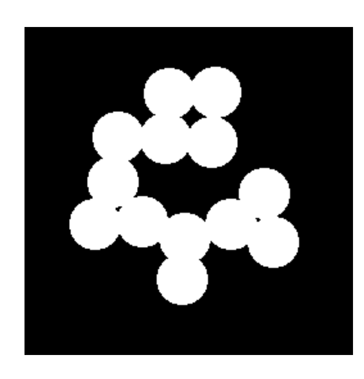
```
eul = bweuler(BW,n)
```

Description `eul = bweuler(BW,n)` returns the Euler number for the binary image `BW`. The return value, `eul`, is a scalar whose value is the total number of objects in the image minus the total number of holes in those objects. The argument `n` can have a value of either 4 or 8, where 4 specifies 4-connected objects and 8 specifies 8-connected objects; if the argument is omitted, it defaults to 8.

Class Support `BW` can be numeric or logical and it must be real, nonsparse and two-dimensional. The return value, `eul`, is of class `double`.

Example

```
BW = imread('circles.tif');
imshow(BW);
```



```
bweuler(BW)

ans =

     2
```

Algorithm `bweuler` computes the Euler number by considering patterns of convexity and concavity in local 2-by-2 neighborhoods. See [2] for a discussion of the algorithm used.

See Also `bwmorph`, `bwperim`

References

[1] Horn, Berthold P. K., *Robot Vision.* New York: McGraw-Hill, 1986. pp. 73-77.

[2] Pratt, William K. *Digital Image Processing*. New York: John Wiley & Sons, Inc., 1991. p. 633.

Purpose

Fill background regions in a binary image

Note This function is obsolete and may be removed in future versions. Use `imfill` instead.

Syntax

```
BW2 = bwfill(BW1,c,r,n)
BW2 = bwfill(BW1,n)
[BW2,idx] = bwfill(...)

BW2 = bwfill(x,y,BW1,xi,yi,n)
[x,y,BW2,idx,xi,yi] = bwfill(...)

BW2 = bwfill(BW1,'holes',n)
[BW2,idx] = bwfill(BW1,'holes',n)
```

Description

`BW2 = bwfill(BW1,c,r,n)` performs a flood-fill operation on the input binary image `BW1`, starting from the pixel (`r`,`c`). If `r` and `c` are equal-length vectors, the fill is performed in parallel from the starting pixels (`r(k)`,`c(k)`). `n` can have a value of either 4 or 8 (the default), where 4 specifies 4-connected foreground and 8 specifies 8-connected foreground. The foreground of `BW1` comprises the `on` pixels (i.e., having value of 1).

`BW2 = bwfill(BW1,n)` displays the image `BW1` on the screen and lets you select the starting points using the mouse. If you omit `BW1`, `bwfill` operates on the image in the current axes. Use normal button clicks to add points. Press **Backspace** or **Delete** to remove the previously selected point. A shift-click, right-click, or double-click selects a final point and then starts the fill; pressing **Return** finishes the selection without adding a point.

`[BW2,idx] = bwfill(...)` returns the linear indices of all pixels filled by `bwfill`.

`BW2 = bwfill(x,y,BW1,xi,yi,n)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system for `BW1`. `xi` and `yi` are scalars or equal-length vectors that specify locations in this coordinate system.

`[x,y,BW2,idx,xi,yi] = bwfill(...)` returns the `XData` and `YData` in `x` and `y`; the output image in `BW2`; linear indices of all filled pixels in `idx`; and the fill starting points in `xi` and `yi`.

`BW2 = bwfill(BW1,'holes',n)` fills the holes in the binary image `BW1`. `bwfill` automatically determines which pixels are in object holes, and then changes the value of those pixels from 0 to 1. `n` defaults to 8 if you omit the argument.

`[BW2,idx] = bwfill(BW1,'holes',n)` returns the linear indices of all pixels filled in by `bwfill`.

If `bwfill` is used with no output arguments, the resulting image is displayed in a new figure.

Remarks

`bwfill` differs from many other binary image operations in that it operates on background pixels, rather than foreground pixels. If the foreground is 8-connected, the background is 4-connected, and vice versa. Note, however, that you specify the connectedness of the *foreground* when you call `bwfill`.

Class Support

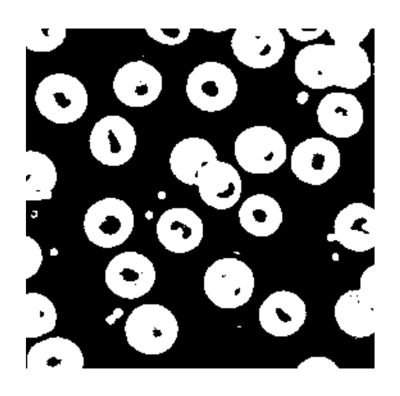
The input image, `BW1`, must be a numeric or logical matrix. The output image, `BW2`, is logical.

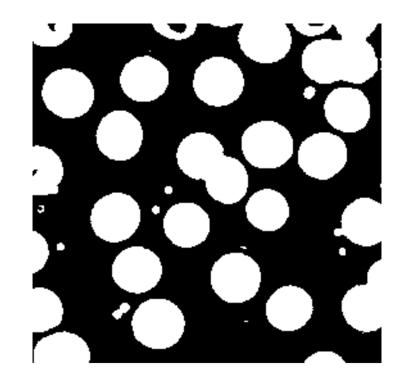
Example

```
BW1 = [1     0     0     0     0     0     0     0
       1     1     1     1     1     0     0     0
       1     0     0     0     1     0     1     0
       1     0     0     0     1     1     1     0
       1     1     1     1     0     1     1     1
       1     0     0     1     1     0     1     0
       1     0     0     0     1     0     1     0
       1     0     0     0     1     1     1     0]

BW2 = bwfill(BW1,3,3,8)
BW2 =
     1     0     0     0     0     0     0     0
     1     1     1     1     1     0     0     0
     1     1     1     1     1     0     1     0
     1     1     1     1     1     1     1     0
     1     1     1     1     0     1     1     1
     1     0     0     1     1     0     1     0
     1     0     0     0     1     0     1     0
     1     0     0     0     1     1     1     0
```

```
I = imread('blood1.tif');
BW3 = ~im2bw(I);
BW4 = bwfill(BW3,'holes');
imshow(BW3)
figure, imshow(BW4)
```





See Also

bwselect, roifill

bwhitmiss

Purpose Binary hit-and-miss operation

Syntax

```
BW2 = bwhitmiss(BW1,SE1,SE2)
BW2 = bwhitmiss(BW1,INTERVAL)
```

Description `BW2 = bwhitmiss(BW1,SE1,SE2)` performs the hit-and-miss operation defined by the structuring elements `SE1` and `SE2`. The hit-and-miss operation preserves pixels whose neighborhoods match the shape of `SE1` and don't match the shape of `SE2`. `SE1` and `SE2` may be flat structuring element objects, created by `strel`, or neighborhood arrays. The neighborhoods of `SE1` and `SE2` should not have any overlapping elements. The syntax `bwhitmiss(BW1,SE1,SE2)` is equivalent to `imerode(BW1,SE1) & imerode(~BW1,SE2)`.

`BW2 = bwhitmiss(BW1,INTERVAL)` performs the hit-and-miss operation defined in terms of a single array, called an *interval*. An interval is an array whose elements can contain either `1`, `0`, or `-1`. The `1`-valued elements make up the domain of `SE1`; the `-1`-valued elements make up the domain of `SE2`; and the `0`-valued elements are ignored. The syntax `bwhitmiss(INTERVAL)` is equivalent to `bwhitmiss(BW1,INTERVAL == 1, INTERVAL == -1)`.

Class support `BW1` can be a logical or numeric array of any dimension, and it must be nonsparse. `BW2` is always a logical array with the same size as `BW1`. `SE1` and `SE2` must be flat `STREL` objects or they must be logical or numeric arrays containing 1s and 0s. `INTERVAL` must be an array containing `1`s, `0`s, and `-1`s.

Example This example performs the hit-and-miss operation on a binary image using an interval.

```
bw = [0 0 0 0 0 0
      0 0 1 1 0 0
      0 1 1 1 1 0
      0 1 1 1 1 0
      0 0 1 1 0 0
      0 0 1 0 0 0]

interval = [0 -1 -1
            1  1 -1
            0  1  0];
```

```
bw2 = bwhitmiss(bw,interval)

bw2 =

     0     0     0     0     0     0
     0     0     0     1     0     0
     0     0     0     0     1     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
```

See Also

`imdilate`, `imerode`, `strel`

bwlabel

Purpose

Label connected components in a binary image

Syntax

```
L = bwlabel(BW,n)
[L,num] = bwlabel(BW,n)
```

Description

`L = bwlabel(BW,n)` returns a matrix `L`, of the same size as `BW`, containing labels for the connected objects in `BW`. `n` can have a value of either 4 or 8, where 4 specifies 4-connected objects and 8 specifies 8-connected objects; if the argument is omitted, it defaults to 8.

The elements of `L` are integer values greater than or equal to 0. The pixels labeled 0 are the background. The pixels labeled 1 make up one object, the pixels labeled 2 make up a second object, and so on.

`[L,num] = bwlabel(BW,n)` returns in `num` the number of connected objects found in `BW`.

Remarks

`bwlabel` supports 2-D inputs only; `bwlabeln` supports inputs of any dimension. In some cases, you might prefer to use `bwlabeln` even for 2-D problems because it can be faster. If you have a 2-D input whose objects are relatively "thick" in the vertical direction, `bwlabel` will probably be faster; otherwise `bwlabeln` will probably be faster.

Class Support

`BW` can be logical or numeric, and it must be real, 2-D, and nonsparse. `L` is of class `double`.

Remarks

You can use the MATLAB `find` function in conjunction with `bwlabel` to return vectors of indices for the pixels that make up a specific object. For example, to return the coordinates for the pixels in object 2

```
[r,c] = find(bwlabel(BW)==2)
```

You can display the output matrix as a pseudo-color indexed image. Each object appears in a different color, so the objects are easier to distinguish than in the original image. See `label2rgb` for more information.

Example

This example illustrates using 4-connected objects. Notice objects 2 and 3; with 8-connected labeling, `bwlabel` would consider these a single object rather than two separate objects.

```
BW = [1     1     1     0     0     0     0     0
      1     1     1     0     1     1     0     0
      1     1     1     0     1     1     0     0
      1     1     1     0     0     0     1     0
      1     1     1     0     0     0     1     0
      1     1     1     0     0     0     1     0
      1     1     1     0     0     1     1     0
      1     1     1     0     0     0     0     0];

L = bwlabel(BW,4)

L =

     1     1     1     0     0     0     0     0
     1     1     1     0     2     2     0     0
     1     1     1     0     2     2     0     0
     1     1     1     0     0     0     3     0
     1     1     1     0     0     0     3     0
     1     1     1     0     0     0     3     0
     1     1     1     0     0     3     3     0
     1     1     1     0     0     0     0     0

[r,c] = find(L==2);
rc = [r c]

rc =

     2     5
     3     5
     2     6
     3     6
```

Algorithm

`bwlabel` uses the general procedure outlined in reference [1], pp. 40-48:

1. Run-length encode the input image.
2. Scan the runs, assigning preliminary labels and recording label equivalences in a local equivalence table.
3. Resolve the equivalence classes.
4. Relabel the runs based on the resolved equivalence classes.

See Also

`bweuler`, `bwlabeln`, `bwselect`, `label2rgb`

Reference

[1] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I.* Addison-Wesley, 1992. pp. 28-48.

Purpose Label connected components in N-D binary image

Syntax

```
L = bwlabeln(BW)
[L,NUM] = bwlabeln(BW)
[L,NUM] = bwlabeln(BW,CONN)
```

Description `L = bwlabeln(BW)` returns a label matrix, `L`, containing labels for the connected components in `BW`. `BW` can have any dimension; `L` is the same size as `BW`. The elements of `L` are integer values greater than or equal to 0. The pixels labeled 0 are the background. The pixels labeled `1` make up one object, the pixels labeled `2` make up a second object, and so on. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions.

`[L,NUM] = bwlabeln(BW)` returns in `NUM` the number of connected objects found in `BW`.

`[L,NUM] = bwlabeln(BW,CONN)` specifies the desired connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may also be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

bwlabeln

Remarks

`bwlabel` supports 2-D inputs only; `bwlabeln` supports inputs of any dimension. In some cases, you might prefer to use `bwlabeln` even for 2-D problems because it can be faster. If you have a 2-D input whose objects are relatively "thick" in the vertical direction, `bwlabel` will probably be faster; otherwise `bwlabeln` will probably be faster.

Class Support

`BW` can be numeric or logical, and it must be real and nonsparse. `L` is of class `double`.

Example

```
BW = cat(3,[1 1 0; 0 0 0; 1 0 0],...
[0 1 0; 0 0 0; 0 1 0],...
[0 1 1; 0 0 0; 0 0 1])

bwlabeln(BW)

ans(:,:,1) =

     1     1     0
     0     0     0
     2     0     0


ans(:,:,2) =

     0     1     0
     0     0     0
     0     2     0


ans(:,:,3) =

     0     1     1
     0     0     0
     0     0     2
```

Algorithm

`bwlabeln` uses the following general procedure:

1. Scan all image pixels, assigning preliminary labels to nonzero pixels and recording label equivalences in a union-find table.

2. Resolve the equivalence classes using the union-find algorithm [1].
3. Relabel the pixels based on the resolved equivalence classes.

See Also

`bwlabel`, `label2rgb`

Reference

[1] Robert Sedgewick, *Algorithms in C*, 3rd ed., Addison-Wesley, 1998, pp. 11-20.

bwmorph

Purpose

Perform morphological operations on binary images

Syntax

```
BW2 = bwmorph(BW1,operation)
BW2 = bwmorph(BW1,operation,n)
```

Description

`BW2 = bwmorph(BW1,operation)` applies a specific morphological operation to the binary image `BW1`.

`BW2 = bwmorph(BW1,operation,n)` applies the operation `n` times. `n` can be `Inf`, in which case the operation is repeated until the image no longer changes.

`operation` is a string that can have one of the values listed below.

'bothat'	'erode'	'shrink'
'bridge'	'fill'	'skel'
'clean'	'hbreak'	'spur'
'close'	'majority'	'thicken'
'diag'	'open'	'thin'
'dilate'	'remove'	'tophat'

`'bothat'` ("bottom hat") performs morphological closing (dilation followed by erosion) and subtracts the original image.

`'bridge'` bridges previously unconnected pixels. For example,

```
1  0  0                 1  0  0
1  0  1     becomes     1  1  1
0  0  1                 0  0  1
```

`'clean'` removes isolated pixels (individual 1's that are surrounded by 0's), such as the center pixel in this pattern.

```
0  0  0
0  1  0
0  0  0
```

`'close'` performs morphological closing (dilation followed by erosion).

`'diag'` uses diagonal fill to eliminate 8-connectivity of the background. For example,

```
0  1  0                 0  1  0
1  0  0     becomes     1  1  0
0  0  0                 0  0  0
```

`'dilate'` performs dilation using the structuring element `ones(3)`.

`'erode'` performs erosion using the structuring element `ones(3)`.

`'fill'` fills isolated interior pixels (individual 0's that are surrounded by 1's), such as the center pixel in this pattern.

```
1  1  1
1  0  1
1  1  1
```

`'hbreak'` removes H-connected pixels. For example,

```
1  1  1                 1  1  1
0  1  0     becomes     0  0  0
1  1  1                 1  1  1
```

`'majority'` sets a pixel to `1` if five or more pixels in its 3-by-3 neighborhood are 1's; otherwise, it sets the pixel to `0`.

`'open'` implements morphological opening (erosion followed by dilation).

`'remove'` removes interior pixels. This option sets a pixel to `0` if all of its 4-connected neighbors are `1`, thus leaving only the boundary pixels `on`.

`'shrink'`, with `n = Inf`, shrinks objects to points. It removes pixels so that objects without holes shrink to a point, and objects with holes shrink to a connected ring halfway between each hole and the outer boundary. This option preserves the Euler number.

`'skel'`, with `n = Inf`, removes pixels on the boundaries of objects but does not allow objects to break apart. The pixels remaining make up the image skeleton. This option preserves the Euler number.

`'spur'` removes spur pixels. For example,

```
0  0  0  0              0  0  0  0
0  0  0  0              0  0  0  0
```

```
0   0   1   0      becomes      0   0   0   0
0   1   0   0                   0   1   0   0
1   1   0   0                   1   1   0   0
```

`'thicken'`, with `n = Inf`, thickens objects by adding pixels to the exterior of objects until doing so would result in previously unconnected objects being 8-connected. This option preserves the Euler number.

`'thin'`, with `n = Inf`, thins objects to lines. It removes pixels so that an object without holes shrinks to a minimally connected stroke, and an object with holes shrinks to a connected ring halfway between each hole and the outer boundary. This option preserves the Euler number.

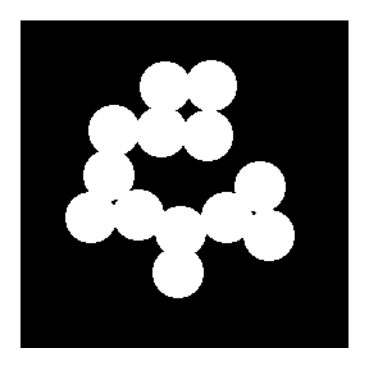
`'tophat'` ("top hat") returns the image minus the morphological opening of the image.

Class Support

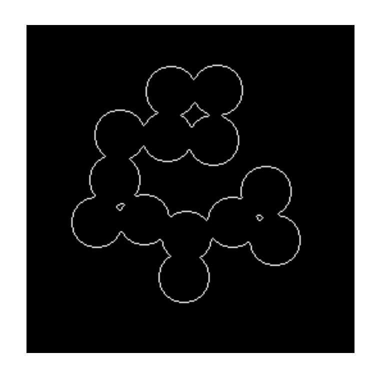
The input image `BW1` can be numeric or logical. It must be 2-D, real and nonsparse. The output image `BW2` is of class `logical`.

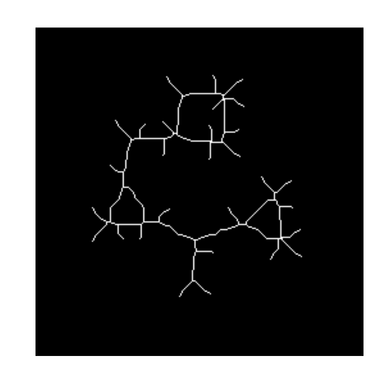
Example

```
BW1 = imread('circles.tif');
imshow(BW1);
```



```
BW2 = bwmorph(BW1,'remove');
BW3 = bwmorph(BW1,'skel',Inf);
imshow(BW2)
figure, imshow(BW3)
```





See Also

`bweuler`, `bwperim`, `dilate`, `erode`

References

[1] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I.* Addison-Wesley, 1992.

[2] Pratt, William K. *Digital Image Processing*. John Wiley & Sons, Inc., 1991.

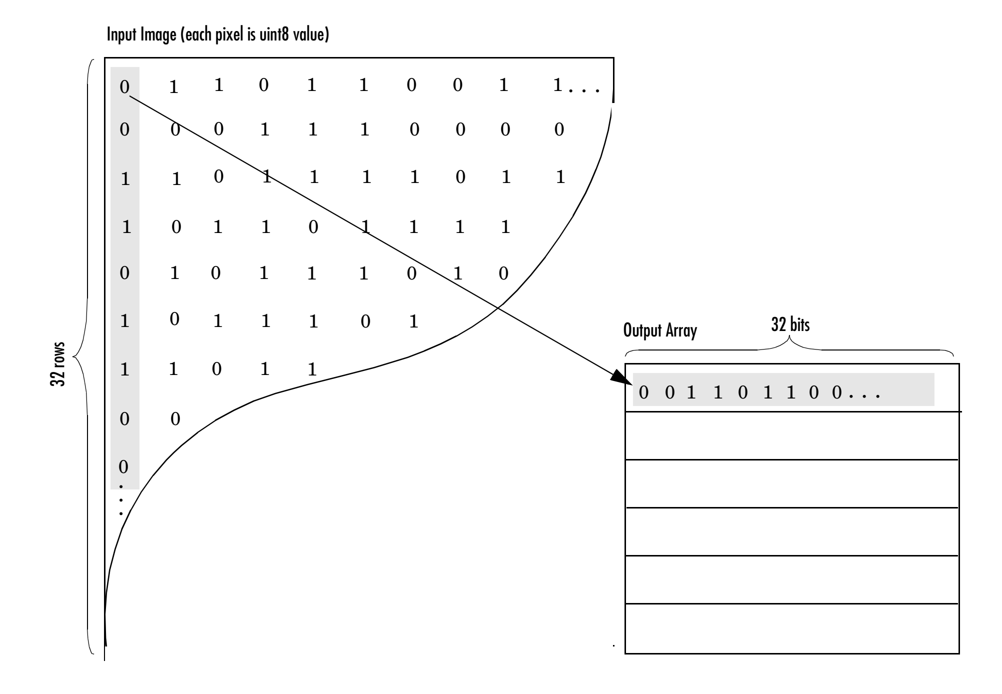
bwpack

Purpose Pack binary image

Syntax `BWP = bwpack(BW)`

Description `BWP = bwpack(BW)` packs the `uint8` binary image `BW` into the `uint32` array `BWP`, which is known as a *packed binary image*. Because each 8-bit pixel value in the binary image has only two possible values, `1` and `0`, `bwpack` can map each pixel to a single bit in the packed output image.

`bwpack` processes the image pixels by column, mapping groups of 32 pixels into the bits of a `uint32` value. The first pixel in the first row corresponds to the least significant bit of the first `uint32` element of the output array. The first pixel in the 32nd input row corresponds to the most significant bit of this same element. The first pixel of the 33rd row corresponds to the least significant bit of the second output element, and so on. If `BW` is `M`-by-`N`, then `BWP` is `ceil(M/32)-by-N`. This figure illustrates how `bwpack` maps the pixels in a binary image to the bits in a packed binary image.



Binary image packing is used to accelerate some binary morphological operations, such as dilation and erosion. If the input to `imdilate` or `imerode` is a packed binary image, the functions use a specialized routine to perform the operation faster.

`bwunpack` is used to unpack packed binary images.

Class Support

`BW` can be logical or numeric, and it must be 2-D, real, and nonsparse. `BWP` is of class `uint32`.

Example

Pack, dilate, and unpack a binary image:

```
bw = imread('text.tif');
bwp = bwpack(bw);
bwp_dilated = imdilate(bwp,ones(3,3),'ispacked );
bw_dilated = bwunpack(bwp_dilated, size(bw,1));
```

See Also

`bwunpack`, `imdilate`, `imerode`

bwperim

Purpose

Find perimeter pixels in binary image

Syntax

```
BW2 = bwperim(BW1)
BW2 = bwperim(BW1,CONN)
```

Description

`BW2 = bwperim(BW1)` returns a binary image containing only the perimeter pixels of objects in the input image `BW1`. A pixel is part of the perimeter if it is nonzero and it is connected to at least one zero-valued pixel. The default connectivity is 4 for two dimensions, 6 for three dimensions, and `conndef(ndims(BW), 'minimal')` for higher dimensions.

`BW2 = bwperim(BW1,CONN)` specifies the desired connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may also be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by-...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

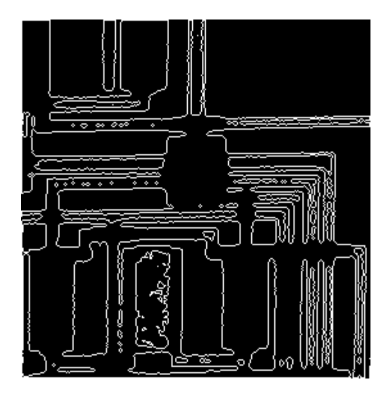
`BW1` must be logical or numeric, and it must be nonsparse. `BW2` is of class `logical`.

Example

```
BW1 = imread('circbw.tif');
BW2 = bwperim(BW1,8);
imshow(BW1)
```

```
figure, imshow(BW2)
```





See Also `bwarea`, `bweuler`, `bwfill`, `conndef`

bwselect

Purpose

Select objects in a binary image

Syntax

```
BW2 = bwselect(BW1,c,r,n)
BW2 = bwselect(BW1,n)
[BW2,idx] = bwselect(...)

BW2 = bwselect(x,y,BW1,xi,yi,n)
[x,y,BW2,idx,xi,yi] = bwselect(...)
```

Description

`BW2 = bwselect(BW1,c,r,n)` returns a binary image containing the objects that overlap the pixel `(r,c)`. `r` and `c` can be scalars or equal-length vectors. If `r` and `c` are vectors, `BW2` contains the sets of objects overlapping with any of the pixels `(r(k),c(k))`. `n` can have a value of either 4 or 8 (the default), where 4 specifies 4-connected objects and 8 specifies 8-connected objects. Objects are connected sets of `on` pixels (i.e., pixels having a value of 1).

`BW2 = bwselect(BW1,n)` displays the image `BW1` on the screen and lets you select the `(r,c)` coordinates using the mouse. If you omit `BW1`, `bwselect` operates on the image in the current axes. Use normal button clicks to add points. Pressing **Backspace** or **Delete** removes the previously selected point. A shift-click, right-click, or double-click selects the final point; pressing **Return** finishes the selection without adding a point.

`[BW2,idx] = bwselect(...)` returns the linear indices of the pixels belonging to the selected objects.

`BW2 = bwselect(x,y,BW1,xi,yi,n)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system for `BW1`. `xi` and `yi` are scalars or equal-length vectors that specify locations in this coordinate system.

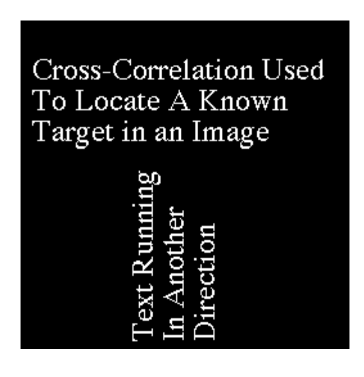
`[x,y,BW2,idx,xi,yi] = bwselect(...)` returns the `XData` and `YData` in `x` and `y`; the output image in `BW2`; linear indices of all the pixels belonging to the selected objects in `idx`; and the specified spatial coordinates in `xi` and `yi`.

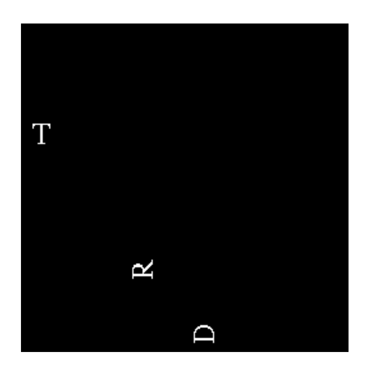
If `bwselect` is called with no output arguments, the resulting image is displayed in a new figure.

Example

```
BW1 = imread('text.tif');
c = [16 90 144];
r = [85 197 247];
```

```
BW2 = bwselect(BW1,c,r,4);
imshow(BW1)
figure, imshow(BW2)
```





Class Support The input image, `BW1`, can be logical or numeric and must be 2-D and nonsparse. The output image, `BW2`, is of class `logical`.

See Also `bwfill`, `bwlabel`, `impixel`, `roipoly`, `roifill`

bwulterode

Purpose

Ultimate erosion

Syntax

```
BW2 = bwulterode(BW)
BW2 = bwulterode(BW,METHOD,CONN)
```

Description

`BW2 = bwulterode(BW)` computes the ultimate erosion of the binary image `BW`. The ultimate erosion of `BW` consists of the regional maxima of the Euclidean distance transform of the complement of `BW`. The default connectivity for computing the regional maxima is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions.

`BW2 = bwulterode(BW,METHOD,CONN)` specifies the distance transform method and the regional maxima connectivity. `METHOD` can be one of the strings `'euclidean'`, `'cityblock'`, `'chessboard'`, or `'quasi-euclidean'`.

`CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by... - by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

`BW` can be numeric or logical and it must be nonsparse. It can have any dimension. The return value, `BW2`, is always a logical array.

Example

```
bw = imread('circles.tif');
imshow(bw), title('Original')
bw2 = bwulterode(bw);
figure, imshow(bw2), title('Ultimate erosion')
```

See Also

`bwdist`, `conndef`, `imregionalmax`

bwunpack

Purpose Unpack binary image

Syntax

```
BW = bwunpack(BWP,M)
```

Description `BW = bwunpack(BWP,M)` unpacks the packed binary image `BWP`. `BWP` is a `uint32` array. When it unpacks `BWP`, `bwunpack` maps the least significant bit of the first row of `BWP` to the first pixel in the first row of `BW`. The most significant bit of the first element of `BWP` maps to the first pixel in the 32nd row of `BW`, and so on. `BW` is `M`-by-`N`, where `N` is the number of columns of `BWP`. If `M` is omitted, its default value is `32*size(BWP,1)`.

Binary image packing is used to accelerate some binary morphological operations, such as dilation and erosion. If the input to `imdilate` or `imerode` is a packed binary image, the functions use a specialized routine to perform the operation faster.

`bwpack` is used to create packed binary images.

Class Support `BWP` is of class `uint32` and must be real, 2-D, and nonsparse. The return value, `BW`, is of class `uint8`.

Example Pack, dilate, and unpack a binary image.

```
bw = imread('text.tif');
bwp = bwpack(bw);
bwp_dilated = imdilate(bwp,ones(3,3),'ispacked');
bw_dilated = bwunpack(bwp_dilated, size(bw,1));
```

See Also `bwpack`, `imdilate`, `imerode`

checkerboard

Purpose Create checkerboard image

Syntax

```
I = checkerboard
I = checkerboard(N)
I = checkerboard(N,P,Q)
```

Description

`I = checkerboard` creates a 8-by-8 square checkerboard image that has four identifiable corners. Each square has 10 pixels per side. The light squares on the left half of the checkerboard are white. The light squares on the right half of the checkerboard are gray.

`I = checkerboard(N)` creates a checkerboard image where each square has `N` pixels per side.

`I = checkerboard(N,P,Q)` creates a rectangular checkerboard, where `P` specifies the number of rows and `Q` specifies the number of columns. Each row and column is made up of tiles. Each tile contains four squares, `N` pixels per side, defined as:

```
TILE = [DARK LIGHT; LIGHT DARK]
```

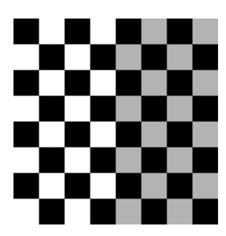


If you omit `Q`, it defaults to `P` and the checkerboard is square

Example

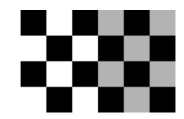
Create a checkerboard where the side of every square is 20 pixels in length.

```
I = checkerboard(20);
imshow(I)
```



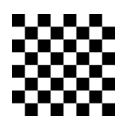
Create a 2-by-3 rectangular checkerboard.

```
J = checkerboard(20,2,3);
figure, imshow(J)
```



Create a black and white checkerboard.

```
K = (checkerboard > 0.5);
figure, imshow(K)
```



See Also `cp2tform`, `imtransform`, `maketform`

Purpose

Rearrange the colors in a colormap

Syntax

```
[Y,newmap] = cmpermute(X,map)
[Y,newmap] = cmpermute(X,map,index)
```

Description

`[Y,newmap] = cmpermute(X,map)` randomly reorders the colors in `map` to produce a new colormap `newmap`. The `cmpermute` function also modifies the values in `X` to maintain correspondence between the indices and the colormap, and returns the result in `Y`. The image `Y` and associated colormap `newmap` produce the same image as `X` and `map`.

`[Y,newmap] = cmpermute(X,map,index)` uses an ordering matrix (such as the second output of `sort`) to define the order of colors in the new colormap.

Class Support

The input image `X` can be of class `uint8` or `double`. `Y` is returned as an array of the same class as `X`.

Example

To arrange a colormap in order by luminance, use

```
ntsc = rgb2ntsc(map);
[dum,index] = sort(ntsc(:,1));
[Y,newmap] = cmpermute(X,map,index);
```

See Also

`randperm`, `sort` in the MATLAB Function Reference

cmunique

Purpose

Find unique colormap colors and the corresponding image

Syntax

```
[Y,newmap] = cmunique(X,map)
[Y,newmap] = cmunique(RGB)
[Y,newmap] = cmunique(I)
```

Description

`[Y,newmap] = cmunique(X,map)` returns the indexed image `Y` and associated colormap `newmap` that produce the same image as `(X,map)` but with the smallest possible colormap. The `cmunique` function removes duplicate rows from the colormap and adjusts the indices in the image matrix accordingly.

`[Y,newmap] = cmunique(RGB)` converts the truecolor image `RGB` to the indexed image `Y` and its associated colormap `newmap`. The return value, `newmap`, is the smallest possible colormap for the image, containing one entry for each unique color in `RGB`. (Note that `newmap` may be very large, because the number of entries can be as many as the number of pixels in `RGB`.)

`[Y,newmap] = cmunique(I)` converts the intensity image `I` to an indexed image `Y` and its associated colormap `newmap`. The return value, `newmap`, is the smallest possible colormap for the image, containing one entry for each unique intensity level in `I`.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`. The class of the output image `Y` is `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `Y` is of class `double`.

See Also

`gray2ind`, `rgb2ind`

Purpose Rearrange matrix columns into blocks

Syntax

```
A = col2im(B,[m n],[mm nn], block_type)
A = col2im(B,[m n],[mm nn])
```

Description `col2im` rearranges matrix columns into blocks. *`block_type`* is a string with one of these values:

- `'distinct'` for `m`-by-`n` distinct blocks
- `'sliding'` for `m`-by-`n` sliding blocks (default)

`A = col2im(B,[m n],[mm nn],'distinct')` rearranges each column of `B` into a distinct `m`-by-`n` block to create the matrix `A` of size `mm`-by-`nn`. If `B = [A11(:) A12(:) A21(:) A22(:)]`, where each column has length `m*n`, then `A = [A11 A12;A21 A22]` where each `Aij` is `m`-by-`n`.

`A = col2im(B,[m n],[mm nn],'sliding')` rearranges the row vector `B` into a matrix of size `(mm-m+1)`-by-`(nn-n+1)`. `B` must be a vector of size 1-by-`(mm-m+1)*(nn-n+1)`. `B` is usually the result of processing the output of `im2col(...,'sliding')` using a column compression function (such as `sum`).

`A = col2im(B,[m n],[mm nn])` uses the default `block_type` of `'sliding'`.

Class Support `B` can be logical or numeric. The return value, `A`, is of the same class as `B`.

See Also `blkproc`, `colfilt`, `im2col`, `nlfilter`

Purpose

Perform neighborhood operations using column-wise functions

Syntax

```
B = colfilt(A,[m n],block_type,fun)
B = colfilt(A,[m n],block_type,fun,P1,P2,...)
B = colfilt(A,[m n],[mblock nblock],block_type,fun,...)
B = colfilt(A,'indexed',...)
```

Description

`colfilt` processes distinct or sliding blocks as columns. `colfilt` can perform similar operations to `blkproc` and `nlfilter`, but often executes much faster.

`B = colfilt(A,[m n],`*`block_type`*`,fun)` processes the image `A` by rearranging each `m`-by-`n` block of `A` into a column of a temporary matrix, and then applying the function `fun` to this matrix. `fun` can be a `function_handle`, created using `@`, or an inline object. `colfilt` zero pads `A`, if necessary.

Before calling `fun`, `colfilt` calls `im2col` to create the temporary matrix. After calling `fun`, `colfilt` rearranges the columns of the matrix back into `m`-by-`n` blocks using `col2im`.

`block_type` is a string with one of these values:

- `'distinct'` for `m`-by-`n` distinct blocks
- `'sliding'` for `m`-by-`n` sliding neighborhoods

`B = colfilt(A,[m n],'distinct',fun)` rearranges each `m`-by-`n` distinct block of `A` into a column in a temporary matrix, and then applies the function `fun` to this matrix. `fun` must return a matrix of the same size as the temporary matrix. `colfilt` then rearranges the columns of the matrix returned by `fun` into `m`-by-`n` distinct blocks.

`B = colfilt(A,[m n],'sliding',fun)` rearranges each `m`-by-`n` sliding neighborhood of `A` into a column in a temporary matrix, and then applies the function `fun` to this matrix. `fun` must return a row vector containing a single value for each column in the temporary matrix. (Column compression functions such as `sum` return the appropriate type of output.) `colfilt` then rearranges the vector returned by `fun` into a matrix of the same size as `A`.

`B = colfilt(A,[m n],`*`block_type`*`,fun,P1,P2,...)` passes the additional parameters `P1,P2,...,` to `fun`. The `colfilt` function calls `fun` using,

```
y = fun(x,P1,P2,...)
```

where x is the temporary matrix before processing, and y is the temporary matrix after processing.

B = colfilt(A,[m n],[mblock nblock],*block_type*,fun,...) processes the matrix A as above, but in blocks of size mblock-by-nblock to save memory. Note that using the [mblock nblock] argument does not change the result of the operation.

B = colfilt(A,'indexed',...) processes A as an indexed image, padding with zeros if the class of A is uint8 or uint16, or ones if the class of A is double.

Class Support

The input image A can be of any class supported by fun. The class of B depends on the class of the output from fun.

Example

This example sets each output pixel to the mean value of the input pixel's 5-by-5 neighborhood.

```
I = imread('tire.tif')
imshow(I)
I2 = uint8(colfilt(I,[5 5],'sliding',@mean));
figure, imshow(I2)
```

See Also

blkproc, col2im, im2col, nlfilter

colorbar

Purpose Display a colorbar

`colorbar` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose Create connectivity array

Syntax

```
CONN = conndef(NUM_DIMS,TYPE)
```

Description

`CONN = conndef(NUM_DIMS,TYPE)` returns the connectivity array defined by `TYPE` for `NUM_DIMS` dimensions. `TYPE` can have either of the values listed in this table.

`'minimal'`	Defines a neighborhood whose neighbors are touching the central element on an (`N`-1)-dimensional surface, for the `N`-dimensional case. See Examples for an illustration.
`'maximal'`	Defines a neighborhood including neighbors that touch the central element in any way; it is `ones(repmat(3,1,NUM_DIMS))`. See Examples for an illustration.

Several Image Processing Toolbox functions use `conndef` to create the default connectivity input argument.

Examples

The minimal connectivity array for two dimensions includes the neighbors touching the central element along a line.

```
conn1 = conndef(2,'minimal')

conn1 =
     0     1     0
     1     1     1
     0     1     0
```

The minimal connectivity array for three dimensions includes all the neighbors touching the central element along a face.

```
conndef(3,'minimal')

ans(:,:,1) =
     0     0     0
     0     1     0
     0     0     0
```

```
ans(:,:,2) =
     0     1     0
     1     1     1
     0     1     0

ans(:,:,3) =
     0     0     0
     0     1     0
     0     0     0
```

The maximal connectivity array for two dimensions includes all the neighbors touching the central element in any way.

```
conn2 = conndef(2,'maximal')

conn2 =
     1     1     1
     1     1     1
     1     1     1
```

Purpose Perform two-dimensional convolution

`conv2` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

convmtx2

Purpose
Compute two-dimensional convolution matrix

Syntax

```
T = convmtx2(H,m,n)
T = convmtx2(H,[m n])
```

Description
`T = convmtx2(H,m,n)` or `T = convmtx2(H,[m n])` returns the convolution matrix `T` for the matrix `H`. If `X` is an `m`-by-`n` matrix, then `reshape(T*X(:),size(H)+[m n]-1)` is the same as `conv2(X,H)`.

Class Support
The inputs are all of class `double`. The output matrix `T` is of class `sparse`. The number of nonzero elements in `T` is no larger than `prod(size(H))*m*n`.

See Also
`conv2`

`convmtx` in the *Signal Processing Toolbox User's Guide*

Purpose

Perform N-dimensional convolution

`convn` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

corr2

Purpose Compute the two-dimensional correlation coefficient between two matrices

Syntax

```
r = corr2(A,B)
```

Description `r = corr2(A,B)` computes the correlation coefficient between `A` and `B`, where `A` and `B` are matrices or vectors of the same size.

Class Support `A` and `B` can be numeric or logical. The return value, `r`, is a scalar `double`.

Algorithm `corr2` computes the correlation coefficient using

$$r = \frac{\sum_m \sum_n (A_{mn} - \bar{A})(B_{mn} - \bar{B})}{\sqrt{\left(\sum_m \sum_n (A_{mn} - \bar{A})^2\right)\left(\sum_m \sum_n (B_{mn} - \bar{B})^2\right)}}$$

where $\bar{A}$ = `mean2(A)`, and $\bar{B}$ = `mean2(B)`.

See Also `std2`

`corrcoef` in the MATLAB Function Reference

Purpose

Infer geometric transformation from control point pairs

Syntax

```
TFORM = cp2tform(input_points,base_points,transformtype)
TFORM = cp2tform(CPSTRUCT,transformtype)
TFORM = cp2tform(input_points,base_points,transformtype,parameter)
TFORM = cp2tform(CPSTRUCT,transformtype,parameter)
[TFORM,input_points,base_points] = cp2tform(CPSTRUCT,...)
[TFORM,input_points,base_points,input_points_bad,base_points_bad]
   = cp2tform(...,'piecewise linear')
```

Description

`TFORM = cp2tform(input_points,base_points,`*`transformtype`*`)` takes pairs of control points and uses them to infer a spatial transformation. The function returns a `TFORM` structure containing the spatial transformation. `input_points` is an m-by-2 `double` matrix containing the x and y coordinates of control points in the image you want to transform. `base_points` is an m-by-2 `double` matrix containing the x and y coordinates of control points specified in the base image. The *`transformtype`* argument specifies the type of transformation you want to infer. For more information about supported types, see Transform Type.

`TFORM = cp2tform(CPSTRUCT,`*`transformtype`*`)` passes a `CPSTRUCT` structure that contains the control point matrices for the input and base images. The Control Point Selection Tool, `cpselect`, creates the `CPSTRUCT`.

`[TFORM,input_points,base_points] = cp2tform(CPSTRUCT,...)` returns the control points that were actually used in `input_points`, and `base_points`. Unmatched and predicted points are not used. For more information, see `cpstruct2pairs`.

Transform Type

`transformtype` specifies the type of spatial transformation to infer. This table lists all of the transformation types supported by `cp2tform`, in order of complexity. (See Algorithms for detailed information about each transform type.) The table includes the minimum number of control point pairs you must select for each type. The `'lwm'` and `'polynomial'` transform types can each

take an optional, additional `parameter`. See the syntax descriptions that follow for details.

Transformation Type	Description	Minimum Control Points	Example
`'linear conformal'`	Use this transformation when shapes in the input image are unchanged, but the image is distorted by some combination of translation, rotation, and scaling. Straight lines remain straight, and parallel lines are still parallel.	2 pairs	
`'affine'`	Use this transformation when shapes in the input image exhibit shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles become parallelograms.	3 pairs	
`'projective'`	Use this transformation when the scene appears "tilted." Straight lines remain straight, but parallel lines converge toward "vanishing points" which may or may not fall within the image.	4 pairs	

'polynomial'	Use this transformation when objects in the image are curved. The higher the order of the polynomial, the better the fit, but the result can contain more curves than the base image.	6 pairs (order 2) 10 pairs (order 3) 16 pairs (order 4)	
'piecewise linear'	Use this transformation when parts of the image appear distorted differently.	4 pairs	
'lwm'	Use this transformation (local weighted mean), when the distortion varies locally and piecewise linear is not sufficient.	6 pairs (12 pairs recommended)	

Note When *transformtype* is 'linear conformal', 'affine', 'projective', or 'polynomial', and input_points and base_points (or CPSTRUCT) have the minimum number of control points needed for a particular transformation, cp2tform finds the coefficients exactly. If input_points and base_points include more than the minimum number of points, cp2tform uses a least squares solution. For more information, see mldivide.

```
TFORM = cp2tform(input_points,base_points,'polynomial',ORDER)
TFORM = cp2tform(CPSTRUCT,'polynomial',order)
```

When 'polynomial' is the transform type, you can optionally specify the order of the polynomial to use. order can be the scalar value 2, 3, or 4. If you omit order, it defaults to 3.

```
TFORM = cp2tform(input_points,base_points,'lwm',N)
TFORM = cp2tform(CPSTRUCT,'lwm',N)
```

When 'lwm' is the transform type, you can optionally specify the number of points, N, used to infer each polynomial. The radius of influence extends out to the furthest control point used to infer that polynomial. The N closest points are used to infer a polynomial of order 2 for each control point pair. If you omit N,

it defaults to 12. N can be as small as 6, but making N small risks generating ill-conditioned polynomials.

```
[TFORM,input_points,base_points,input_points_bad,base_points_bad]=
 cp2tform(input_points,base_points,'piecewise linear')
[TFORM,input_points,base_points,input_points_bad,base_points_bad]=
 cp2tform(CPSTRUCT,'piecewise linear')
```

When 'piecewise linear' is the transform type, cp2tform can optionally return the control points that were actually used in input_points and base_points, and return two arrays, input_points_bad and base_points_bad, that contain control points that were eliminated because they were middle vertices of degenerate fold-over triangles.

Algorithms

cp2tform uses the following general procedure:

1. Use valid pairs of control points to infer a spatial transformation or an inverse-mapping from output space (x,y) to input space (u,v) according to transformtype.
2. Return TFORM structure containing spatial transformation.

The procedure varies depending on the transformtype.

Linear Conformal

Linear conformal transformations may include a rotation, a scaling, and a translation. Shapes and angles are preserved. Parallel lines remain parallel. Straight lines remain straight.

Let:

```
sc = scale*cos(angle)
ss = scale*sin(angle)

[u v] = [x y 1] * [ sc -ss
                    ss  sc
                    tx  ty]
```

Solve for sc, ss, tx, ty.

```
t_lc = cp2tform(input_points,base_points,'linear conformal');
```

The coefficients of the inverse mapping are stored in t_lc.tdata.Tinv.

Since linear conformal transformations are a subset of affine transformations, `t_lc.forward_fcn` is `@affine_fwd` and `t_lc.inverse_fcn` is `@affine_inv`.

At least two control-point pairs are needed to solve for the four unknown coefficients.

Affine

In an affine transformation, the `x` and `y` dimensions can be scaled or sheared independently and there may be a translation. Parallel lines remain parallel. Straight lines remain straight. Linear conformal transformations are a subset of affine transformations.

For an affine transformation:

```
[u v] = [x y 1] * Tinv
```

`Tinv` is a 3-by-2 matrix. Solve for the 6 elements of `Tinv`.

```
t_affine = cp2tform(input_points,base_points,'affine');
```

The coefficients of the inverse mapping are stored in `t_affine.tdata.Tinv`.

At least 3 control-point pairs are needed to solve for the 6 unknown coefficients.

Projective

In a projective transformation, quadrilaterals map to quadrilaterals. Straight lines remain straight. Affine transformations are a subset of projective transformations.

For a projective transformation:

```
[up vp wp] = [x y w] * Tinv
```

where

```
u = up/wp
v = vp/wp
```

`Tinv` is a 3-by-3 matrix.

Assuming

```
Tinv = [ A D G;
         B E H;
```

```
          C F I ];

u = (Ax + By + C)/(Gx + Hy + I)
v = (Dx + Ey + F)/(Gx + Hy + I)
```

Solve for the 9 elements of `Tinv`.

```
t_proj = cp2tform(input_points,base_points,'projective');
```

The coefficients of the inverse mapping are stored in `t_proj.tdata.Tinv`.

At least 4 control-point pairs are needed to solve for the 9 unknown coefficients.

Polynomial

In a polynomial transformation, polynomial functions of `x` and `y` determine the mapping.

Second-Order Polynomials

For a second-order polynomial transformation:

```
[u v] = [1  x  y  x*y  x^2  y^2] * Tinv
```

Both `u` and `v` are second-order polynomials of `x` and `y`. Each second-order polynomial has six terms. To specify all coefficients, `Tinv` has size 6-by-2.

```
t_poly_ord2 =
cp2tform(input_points,base_points,'polynomial');
```

The coefficients of the inverse mapping are stored in `t_poly_ord2.tdata`.

At least 6 control-point pairs are needed to solve for the 12 unknown coefficients.

Third-Order Polynomials

For a third-order polynomial transformation:

```
[u v] = [1  x  y  x*y  x^2  y^2  y*x^2  x*y^2  x^3  y^3] * Tinv
```

Both u and v are third-order polynomials of x and y. Each third-order polynomial has ten terms. To specify all coefficients, Tinv has size 10-by-2.

```
t_poly_ord3 = cp2tform(input_points, base_points,
                        'polynomial',3);
```

The coefficients of the inverse mapping are stored in t_poly_ord3.tdata.

At least 10 control-point pairs are needed to solve for the 20 unknown coefficients.

Fourth-Order Polynomials

For a fourth-order polynomial transformation:

```
[u v] = [1  x  y  x*y  x^2  y^2  y*x^2  x*y^2  x^3  y^3] * Tinv
```

Both u and v are fourth-order polynomials of x and y. Each fourth-order polynomial has fifteen terms. To specify all coefficients, Tinv has size 15-by-2.

```
t_poly_ord4 = cp2tform(input_points, base_points,
                         'polynomial',4);
```

The coefficients of the inverse mapping are stored in t_poly_ord4.tdata.

At least 15 control-point pairs are needed to solve for the 30 unknown coefficients.

Piecewise Linear

In a piecewise linear transformation, linear (affine) transformations are applied separately to each triangular region of the image [1].

1. Find a Delaunay triangulation of the base control points.
2. Using the 3 vertices of each triangle, infer an affine mapping from base to input coordinates.

Note At least 4 control-point pairs are needed. Four pairs result in two triangles with distinct mappings.

Local Weighted Mean

For each control point in `base_points`:

1. Find the `N` closest control points.
2. Use these `N` points and their corresponding points in `input_points` to infer a second order polynomial.
3. Calculate the radius of influence of this polynomial as the distance from the center control point to the furthest point used to infer the polynomial (using base_points). [2]

Note At least 6 control-point pairs are needed to solve for the second-order polynomial. Ill-conditioned polynomials may result if too few pairs are used.

Example

```
I = checkerboard;
J = imrotate(I,30);
base_points = [11 11; 41 71];
input_points = [14 44; 70 81];
cpselect(J,I,input_points,base_points);

t = cp2tform(input_points,base_points,'linear conformal');
```

To recover angle and scale:

```
ss = t.tdata.Tinv(2,1); % ss = scale * sin(angle)
sc = t.tdata.Tinv(1,1); % sc = scale * cos(angle)
angle = atan2(ss,sc)*180/pi
scale = sqrt(ss*ss + sc*sc)
```

See Also `cpcorr`, `cpselect`, `cpstruct2pairs`, `imtransform`

References

[1] Ardeshir Goshtasby, Piecewise linear mapping functions for image registration, *Pattern Recognition*, Vol 19, pp. 459-466, 1986.

[2] Ardeshir Goshtasby, Image registration by local approximation methods, *Image and Vision Computing*, Vol 6, p. 255-261, 1988.

cpcorr

Purpose Tune control point locations using cross correlation

Syntax

```
input_points = cpcorr(input_points_in,base_points_in,input,base)
```

Description

`input_points = cpcorr(input_points_in,base_points_in,input,base)` uses normalized cross-correlation to adjust each pair of control points specified in `input_points_in` and `base_points_in`.

`input_points_in` must be an `M`-by-2 `double` matrix containing the coordinates of control points in the `input` image. `base_points_in` is an `M`-by-2 `double` matrix containing the coordinates of control points in the `base` image.

`cpcorr` returns the adjusted control points in `input_points`, a `double` matrix the same size as `input_points_in`. If `cpcorr` cannot correlate a pair of control points, `input_points` contains the same coordinates as `input_points_in` for that pair.

`cpcorr` only moves the position of a control point by up to 4 pixels. Adjusted coordinates are accurate to one tenth of a pixel. `cpcorr` is designed to get subpixel accuracy from the image content and coarse control point selection.

Note `input` and `base` images must have the same scale for `cpcorr` to be effective.

`cpcorr` cannot adjust a point if any of the following occur:

- Points are too near the edge of either image
- Regions of images around points contain `Inf` or `NaN`
- Region around a point in input image has zero standard deviation
- Regions of images around points are poorly correlated

Class Support The images, `input` and `base`, can be of class `logical`, `uint8`, `uint16`, or `double` and must contain finite values. The control point pairs are of class `double`.

Algorithm `cpcorr` uses the following general procedure.

For each control point pair:

1. Extract an 11-by-11 template around the `input` control point and a 21-by-21 region around the `base` control point.
2. Calculate the normalized cross-correlation of the template with the region.
3. Find the absolute peak of the cross-correlation matrix.
4. Use the position of the peak to adjust the coordinates of the `input` control point.

Example

This example uses `cpcorr` to fine-tune control points selected in an image. Note the difference in the values of the `input_points` matrix and the `input_points_adj` matrix.

```
input = imread('lily.tif');
base = imread('flowers.tif');
input_points = [127 93; 74 59];
base_points = [323 195; 269 161];
input_points_adj = cpcorr(input_points,base_points,...
                          input(:,:,1),base(:,:,1))
input_points_adj =

  126.0000   94.0000
   72.1000   60.0000
```

See Also

`cp2tform`, `cpselect`, `imtransform`, `normxcorr2`

cpselect

Purpose

Control Point Selection Tool

Syntax

```
cpselect(input,base)
cpselect(input,base,CPSTRUCT_IN )
cpselect(input,base,XYINPUT_IN,XYBASE_IN)
H = cpselect(input,base,...)
```

Description

`cpselect(input,base)` starts the Control Point Selection Tool, a graphical user interface that enables you to select control points in two related images. `input` is the image that needs to be warped to bring it into the coordinate system of the `base` image. `input` and `base` can be either variables that contain images or strings that identify files containing grayscale images. The Control Point Selection Tool returns the control points in a `CPSTRUCT` structure. (For more information, see "Using the Control Point Selection Tool" in Chapter 5.)

`cpselect(input,base,CPSTRUCT_IN)` starts `cpselect` with an initial set of control points that are stored in `CPSTRUCT_IN`. This syntax allows you to restart `cpselect` with the state of control points previously saved in `CPSTRUCT_IN`.

`cpselect(input,base,xyinput_in,xybase_in)` starts `cpselect` with a set of initial pairs of control points. `xyinput_in` and `xybase_in` are m-by-2 matrices that store the `input` and `base` coordinates, respectively.

`H = cpselect(input,base,...)` returns a handle `H` to the tool. You can use the `dispose(H)` or `H.dispose` syntaxes to close the tool from the command line.

Class Support

The input images can be of class `uint8`, `uint16`, `double`, or `logical`.

Algorithm

`cpselect` uses the following general procedure for control point prediction.

1. Find all valid pairs of control points.
2. Infer a spatial transformation between `input` and `base` control points using method that depends on the number of valid pairs as follows:

2 pairs	Linear conformal
3 pairs	Affine
4 or more pairs	Projective

3. Apply spatial transformation to the new point to generate the predicted point.
4. Display predicted point.

Example

Start tool with saved images.

```
aerial = imread('westconcordaerial.png');
cpselect(aerial(:,:,1),'westconcordorthophoto.png')
```

Start tool with workspace images and points.

```
I = checkerboard;
J = imrotate(I,30);
base_points = [11 11; 41 71];
input_points = [14 44; 70 81];
cpselect(J,I,input_points,base_points);
```

See Also

`cpcorr`, `cp2tform`, `cpstruct2pairs`, `imtransform`

cpstruct2pairs

Purpose Convert CPSTRUCT to valid pairs of control points

Syntax

```
[input_points, base_points] = cpstruct2pairs(CPSTRUCT)
```

Description [input_points, base_points] = cpstruct2pairs(CPSTRUCT) takes a CPSTRUCT (produced by cpselect) and returns the arrays of coordinates of valid control point pairs in input_points and base_points. cpstruct2pairs eliminates unmatched points and predicted points.

Example Start the Control Point Selection Tool, cpselect.

```
cpselect('lily.tif','flowers.tif')
```

Using cpselect, pick control points in the images. Select **Save To Workspace** from the **File** menu to save the points to the workspace. On the **Save** dialog box, check the **Structure with all points** checkbox and uncheck **Input points** and **Base points**. Click **OK**. Use cpstruct2pairs to extract the input and base points from the CPSTRUCT.

```
[input_points,base_points] = cpstruct2pairs(cpstruct);
```

See Also cp2tform, cpselect, imtransform

Purpose

Compute two-dimensional discrete cosine transform

Syntax

```
B = dct2(A)
B = dct2(A,m,n)
B = dct2(A,[m n])
```

Description

`B = dct2(A)` returns the two-dimensional discrete cosine transform of `A`. The matrix `B` is the same size as `A` and contains the discrete cosine transform coefficients $B(k_1,k_2)$.

`B = dct2(A,m,n)` or `B = dct2(A,[m n])` pads the matrix `A` with zeros to size `m`-by-`n` before transforming. If `m` or `n` is smaller than the corresponding dimension of `A`, `dct2` truncates `A`.

Class Support

`A` can be numeric or logical. The returned matrix, `B`, is of class `double`.

Algorithm

The discrete cosine transform (DCT) is closely related to the discrete Fourier transform. It is a separable, linear transformation; that is, the two-dimensional transform is equivalent to a one-dimensional DCT performed along a single dimension followed by a one-dimensional DCT in the other dimension. The definition of the two-dimensional DCT for an input image `A` and output image `B` is

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{array}{l} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{array}$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases} \qquad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

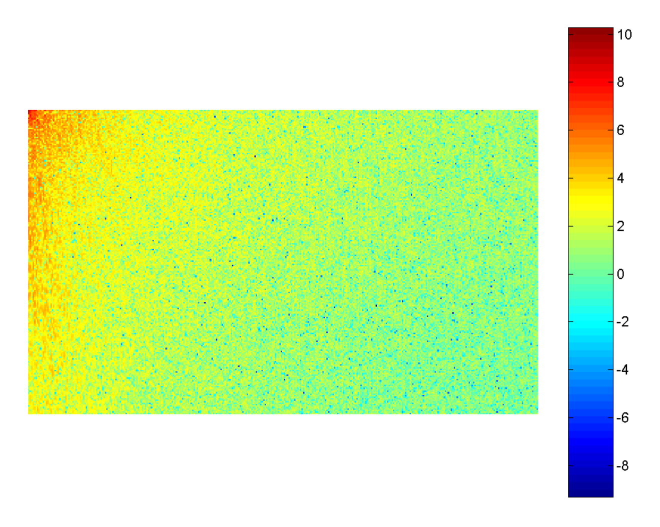
where M and N are the row and column size of `A`, respectively. If you apply the DCT to real data, the result is also real. The DCT tends to concentrate information, making it useful for image compression applications.

This transform can be inverted using `idct2`.

Example

The commands below compute the discrete cosine transform for the `autumn` image. Notice that most of the energy is in the upper left corner.

```
RGB = imread('autumn.tif');
I = rgb2gray(RGB);
J = dct2(I);
imshow(log(abs(J)),[]), colormap(jet(64)), colorbar
```



Now set values less than magnitude 10 in the DCT matrix to zero, and then reconstruct the image using the inverse DCT function `idct2`.

```
J(abs(J) < 10) = 0;
K = idct2(J)/255;
imshow(K)
```



See Also

`fft2`, `idct2`, `ifft2`

References

[1] Jain, Anil K. *Fundamentals of Digital Image Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989. pp. 150-153.

[2] Pennebaker, William B., and Joan L. Mitchell. *JPEG: Still Image Data Compression Standard.* Van Nostrand Reinhold, 1993.

dctmtx

Purpose

Compute discrete cosine transform matrix

Syntax

```
D = dctmtx(n)
```

Description

D = dctmtx(n) returns the n-by-n DCT (discrete cosine transform) matrix. D*A is the DCT of the columns of A and D'*A is the inverse DCT of the columns of A (when A is n-by-n).

Class Support

n is an integer scalar of class double. D is returned as a matrix of class double.

Remarks

If A is square, the two-dimensional DCT of A can be computed as D*A*D'. This computation is sometimes faster than using dct2, especially if you are computing a large number of small DCTs, because D needs to be determined only once.

For example, in JPEG compression, the DCT of each 8-by-8 block is computed. To perform this computation, use dctmtx to determine D, and then calculate each DCT using D*A*D' (where A is each 8-by-8 block). This is faster than calling dct2 for each individual block.

See Also

dct2

Purpose

Restore image using the blind deconvolution algorithm

Syntax

```
[J,PSF] = deconvblind(I,INITPSF)
[J,PSF] = deconvblind(I,INITPSF,NUMIT)
[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR)
[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT)
[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT,READOUT)
[J,PSF] = deconvblind(...,FUN,P1,P2,...,PN)
```

Description

`[J,PSF] = deconvblind(I,INITPSF)` deconvolves image `I` using the maximum likelihood algorithm, returning both the deblurred image, `J`, and a restored point-spread function, `PSF`. The input array, `I`, and your initial guess at the PSF, `INITPSF`, can be numeric arrays or cell arrays. (Use cell arrays when you want to be able to perform additional deconvolutions that start where your initial deconvolution finished. See Resuming Deconvolution for more information.) The restored `PSF` is a positive array that is the same size as `INITPSF`, normalized so its sum adds up to 1.

Note The `PSF` restoration is affected strongly by the size of the initial guess, `INITPSF`, and less by the values it contains. For this reason, specify an array of ones as your `INITPSF`.

To improve the restoration, `deconvblind` supports several optional parameters, described below. Use `[]` as a place holder if you do not specify an intermediate parameter.

`[J,PSF] = deconvblind(I,INITPSF,NUMIT)` specifies the number of iterations (default is 10).

`[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR)` specifies the threshold deviation of the resulting image from the input image `I` (in terms of the standard deviation of Poisson noise) below which damping occurs. The iterations are suppressed for the pixels that deviate within the `DAMPAR` value from their original value. This suppresses the noise generation in such pixels, preserving necessary image details elsewhere. The default value is `0` (no damping).

`[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT)` specifies which pixels in the input image, `I`, are considered in the restoration. By default, `WEIGHT` is a unit array, the same size as the input image. You can assign a value between 0.0 and 1.0 to elements in the `WEIGHT` array. The value of an element in the `WEIGHT` array determines how much the pixel at the corresponding position in the input image is considered. For example, to exclude a pixel from consideration, assign it a value of `0` in the `WEIGHT` array. You can adjust the weight value assigned to each pixel according to the amount of flat-field correction.

`[J,PSF] = deconvblind(I,INITPSF,NUMIT,DAMPAR,WEIGHT,READOUT)`, where `READOUT` is an array (or a value) corresponding to the additive noise (e.g., background, foreground noise) and the variance of the read-out camera noise. `READOUT` has to be in the units of the image. The default value is `0`.

`[J,PSF] = deconvblind(...,FUN,P1,P2,...,PN)`, where `FUN` is a function describing additional constraints on the PSF. There are four ways to specify `FUN`:

- Function handle (@)
- Inline object
- String containing function name
- String containing a MATLAB expression

`FUN` is called at the end of each iteration. `FUN` must accept the `PSF` as its first argument and can accept additional parameters `P1`, `P2,...`, `PN`. The `FUN` function should return one argument, `PSF`, that is the same size as the original PSF and which satisfies the positivity and normalization constraints.

Note The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing use `I = edgetaper(I,PSF)` prior calling `deconvblind`.

Resuming Deconvolution

You can use `deconvblind` to perform a deconvolution that starts where a previous deconvolution stopped. To use this feature, pass the input image, `I`, and the initial guess at the PSF, `INITPSF`, as cell arrays: `{I}` and `{INITPSF}`. When you do, the `deconvblind` function returns the output image `J` and the

restored point spread function, PSF, as cell arrays, which can then be passed as the input arrays into the next deconvblind call. The output cell array, J, contains four elements:

J{1} contains I, the original image

J{2} contains the result of the last iteration

J{3} contains the result of the next-to-last iteration

J{4} is an array generated by the iterative algorithm

Class Support

I can be of class uint8, uint16, or double. The DAMPAR and READOUT arguments have to be of the same class as the input image. Other inputs have to be of class double. The output image, J, or the first array of the output cell array, is of the same class as the input image.

Example

```
I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .0001;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
WT = zeros(size(I));
WT(5:end-4,5:end-4) = 1;
INITPSF = ones(size(PSF));
FUN = inline('PSF + P1','PSF','P1');
[J P]= deconvblind(BlurredNoisy,INITPSF,20,10*sqrt(V),WT,FUN,0);

subplot(221);imshow(BlurredNoisy);
title('A = Blurred and Noisy');
subplot(222);imshow(PSF,[]);
title('True PSF');
subplot(223);imshow(J);
title('Deblured Image');
subplot(224);imshow(P,[]);
title('Recovered PSF');
```

See Also

deconvlucy, deconvreg, deconvwnr, otf2psf, padarray, psf2otf

deconvlucy

Purpose

Restore image using the Lucy-Richardson algorithm

Syntax

```
J = deconvlucy(I,PSF)
J = deconvlucy(I,PSF,NUMIT)
J = deconvlucy(I,PSF,NUMIT,DAMPAR)
J = deconvlucy(I,PSF,NUMIT,DAMPAR,READOUT)
J = deconvlucy(I,PSF,NUMIT,DAMPAR,READOUT,WEIGHT)
deconvlucy(I,PSF,NUMIT,DAMPAR,READOUT,WEIGHT,SUBSMPL)
```

Description

`J = deconvlucy(I,PSF)` restores image `I`, degraded by convolution with a point-spread function, `PSF`, and possibly by additive noise. The algorithm is based on maximizing the likelihood of the resulting image `J` being an instance of the original image `I` under Poisson statistics. The input array, `I`, can be a numeric array or cell array. (Use a cell array when you want to be able to perform additional deconvolutions that start where your initial deconvolution finished. See Resuming Deconvolution for more information.)

To improve the restoration, `deconvlucy` supports several optional parameters. Use `[]` as a place holder if you do not specify an intermediate parameter.

`J = deconvlucy(I,PSF,NUMIT)` specifies the number of iterations the `deconvlucy` function performs. If this value is not specified, the default is 10.

`J = deconvlucy(I,PSF,NUMIT,DAMPAR)` specifies the threshold deviation of the resulting image from the image `I` (in terms of the standard deviation of Poisson noise), below which damping occurs. Iterations are suppressed for pixels that deviate beyond the `DAMPAR` value from their original value. This suppresses the noise generation in such pixels, preserving necessary image details elsewhere. The default value is 0 (no damping).

`J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT)` specifies the weight to be assigned to each pixel to reflect its recording quality in the camera. A bad pixel is excluded from the solution by assigning it zero weight value. Instead of giving a weight of unity for good pixels, one could adjust their weight according to the amount of flat-field correction. The default is a unit array of the same size as input image `I`.

`J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT,READOUT)` specifies a value corresponding to the additive noise (e.g., background, foreground noise) and

the variance of the read-out camera noise. `READOUT` has to be in the units of the image. The default value is 0.

`J = deconvlucy(I,PSF,NUMIT,DAMPAR,WEIGHT,READOUT,SUBSMPL)`, where `SUBSMPL` denotes subsampling and is used when the `PSF` is given on a grid that is `SUBSMPL` times finer than the image. The default value is 1.

Note The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing use `I = edgetaper(I,PSF)` prior calling `deconvlucy`.

Resuming Deconvolution

If input `I` is a cell array, the output `J` becomes a cell array of size 1-by-4, where

- `J{1}` contains `I`, the original image
- `J{2}` contains the result of the last iteration
- `J{3}` contains the result of the next-to-last iteration
- `J{4}` is an array generated by the iterative algorithm

The input cell array could contain one numerical array (the blurred image), or four numerical arrays if it was the output from the previous run of `deconvlucy`.

Class Support

`I` can be of class `uint8`, `uint16`, or `double`. The `DAMPAR` and `READOUT` arguments have to be of the same class as the input image. Other inputs have to be of class `double`. Output image (or the first array of the output cell) is of the same class as the input image.

Example

```
I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .0001;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
WT = zeros(size(I));
WT(5:end-4,5:end-4) = 1;
J1 = deconvlucy(BlurredNoisy,PSF);
J2 = deconvlucy(BlurredNoisy,PSF,20,sqrt(V));
J3 = deconvlucy(BlurredNoisy,PSF,20,sqrt(V),[],WT);

subplot(221);imshow(BlurredNoisy);
```

```
title('A = Blurred and Noisy');
subplot(222);imshow(J1);
title('deconvlucy(A,PSF)');
subplot(223);imshow(J2);
title('deconvlucy(A,PSF,NI,DP)');
subplot(224);imshow(J3);
title('deconvlucy(A,PSF,NI,DP,[],WT)');
```

See Also

deconvblind, deconvreg, deconvwnr, otf2psf, padarray, psf2otf

Purpose

Restore image using a regularized filter

Syntax

```
J = deconvreg(I,PSF)
J = deconvreg(I,PSF,NOISEPOWER)
J = deconvreg(I,PSF,NOISEPOWER,LRANGE)
J = deconvreg(I,PSF,NOISEPOWER,LRANGE,REGOP)
[J, LAGRA] = deconvreg(I,PSF,...)
```

Description

`J = deconvreg(I,PSF)` restores image `I` that was degraded by convolution with a point-spread function PSF and possibly by additive noise. The algorithm is a constrained optimum in a sense of least square error between the estimated and the true images under requirement of preserving image smoothness.

`J = deconvreg(I,PSF,NOISEPOWER)`, where `NOISEPOWER` is the additive noise power. The default value is 0.

`J = deconvreg(I,PSF,NOISEPOWER,LRANGE)`, where `LRANGE` is a vector specifying range where the search for the optimal solution is performed. The algorithm finds an optimal Lagrange multiplier, `LAGRA`, within the `LRANGE` range. If `LRANGE` is a scalar, the algorithm assumes that `LAGRA` is given and equal to `LRANGE`; the `NP` value is then ignored. The default range is between [1e-9 and 1e9].

`J = deconvreg(I,PSF,NOISEPOWER,LRANGE,REGOP)`, where `REGOP` is the regularization operator to constrain the deconvolution. The default regularization operator is the Laplacian operator, to retain the image smoothness. The `REGOP` array dimensions must not exceed the image dimensions, any nonsingleton dimensions must correspond to the nonsingleton dimensions of `PSF`.

`[J, LAGRA] = deconvreg(I,PSF,...)` outputs the value of the Lagrange multiplier, `LAGRA`, in addition to the restored image `J`.

Note The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, process the image with the `edgetaper` function prior to calling the `deconvreg` function; for example, `I = edgetaper(I,PSF)`.

Class Support

I can be of class uint8, uint16, or double. Other inputs have to be of class double. J is of the same class as I.

Example

```
I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .01;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
NOISEPOWER = V*prod(size(I));
[J LAGRA] = deconvreg(BlurredNoisy,PSF,NOISEPOWER);

subplot(221); imshow(BlurredNoisy);
title('A = Blurred and Noisy');
subplot(222); imshow(J);
title('[J LAGRA] = deconvreg(A,PSF,NP)');
subplot(223); imshow(deconvreg(BlurredNoisy,PSF,[],LAGRA/10));
title('deconvreg(A,PSF,[],0.1*LAGRA)');
subplot(224); imshow(deconvreg(BlurredNoisy,PSF,[],LAGRA*10));
title('deconvreg(A,PSF,[],10*LAGRA)');
```

See Also

deconvblind, deconvlucy, deconvwnr, otf2psf, padarray, psf2otf

Purpose

Restore image using the Wiener filter

Syntax

```
J = deconvwnr(I,PSF)
J = deconvwnr(I,PSF,NSR)
J = deconvwnr(I,PSF,NCORR,ICORR)
```

Description

`J = deconvwnr(I,PSF)` restores image `I` that was degraded by convolution with a point-spread function, `PSF`, and possibly by additive noise. The algorithm is optimal in a sense of least mean square error between the estimated and the true image, and uses the correlation matrixes of image and noise. In the absence of noise, the Weiner filter reduces to the ideal inverse filter.

`J = deconvwnr(I,PSF,NSR)`, where `NSR` is the noise-to-signal power ratio. `NSR` could be a scalar or an array of the same size as `I`. The default value is 0.

`J = deconvwnr(I,PSF,NCORR,ICORR)`, where `NCORR` and `ICORR` are the autocorrelation functions of the noise and the original image, respectively. `NCORR` and `ICORR` could be of any size or dimension not exceeding the original image. An N-dimensional `NCORR` or `ICORR` array corresponds to the autocorrelation within each dimension. A vector `NCORR` or `ICORR` represents an autocorrelation function in first dimension if `PSF` is a vector. If `PSF` is an array, the 1-D autocorrelation function is extrapolated by symmetry to all nonsingleton dimensions of `PSF`. A scalar `NCORR` or `ICORR` represents the power of the noise or the image.

Note The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, process the image with the `edgetaper` function prior to calling the `deconvwnr` function; for example, `I = edgetaper(I,PSF)`

Class Support

`I` can be of class `uint8`, `uint16`, or `double`. Other inputs have to be of class `double`. `J` is of the same class as `I`.

Example

```
I = checkerboard(8);
noise = 0.1*randn(size(I));
PSF = fspecial('motion',21,11);
Blurred = imfilter(I,PSF,'circular');
```

```
BlurredNoisy = im2uint8(Blurred + noise);

NSR = sum(noise(:).^2)/sum(I(:).^2);% noise-to-power ratio

NP = abs(fftn(noise)).^2;% noise power
NPOW = sum(NP(:))/prod(size(noise));
NCORR = fftshift(real(ifftn(NP)));% noise autocorrelation
function, centered

IP = abs(fftn(I)).^2;% original image power
IPOW = sum(IP(:))/prod(size(I));
ICORR = fftshift(real(ifftn(IP)));% image autocorrelation
function, centered
ICORR1 = ICORR(:,ceil(size(I,1)/2));

NSR = NPOW/IPOW;
subplot(221);imshow(BlurredNoisy,[]);
title('A = Blurred and Noisy');
subplot(222);imshow(deconvwnr(BlurredNoisy,PSF,NSR),[]);
title('deconvwnr(A,PSF,NSR)');
subplot(223);imshow(deconvwnr(BlurredNoisy,PSF,NCORR,ICORR),[]);
title('deconvwnr(A,PSF,NCORR,ICORR)');
subplot(224);imshow(deconvwnr(BlurredNoisy,PSF,NPOW,ICORR1),[]);
title('deconvwnr(A,PSF,NPOW,ICORR_1_D)');
```

See Also

deconvblind, deconvlucy, deconvreg, otf2psf, padarray, psf2otf

Purpose Read metadata from a DICOM message

Syntax

```
info = dicominfo(filename)
info = dicominfo(filename,'dictionary', D)
```

Description `info = dicominfo(filename)` reads the metadata from the compliant Digital Imaging and Communications in Medicine (DICOM) file specified in the string, `filename`.

`info = dicominfo(filename,'dictionary', D)` uses the data dictionary file given in the string `D` to read the DICOM message. The file in `D` must be on the MATLAB search path. The default dictionary file is `dicom-dict.txt`.

Examples

```
info = dicominfo('CT-MONO2-16-ankle.dcm')

info =
                        Filename: [1x47 char]
                     FileModDate: '24-Dec-2000 19:54:47'
                        FileSize: 525436
                          Format: 'DICOM'
                   FormatVersion: 3
                           Width: 512
                          Height: 512
                        BitDepth: 16
                       ColorType: 'grayscale'
                           .
                           .
                           .
```

See Also `dicomread`, `dicomwrite`

dicomread

Purpose Read a DICOM image

Syntax

```
X = dicomread(filename)
X = dicomread(info)
[X,map] = dicomread(...)
[X,map,alpha] = dicomread(...)
[X,map,alpha,overlays] = dicomread(...)
```

Description

`X = dicomread(filename)` reads the image data from the compliant Digital Imaging and Communications in Medicine (DICOM) file, `filename`. For single-frame grayscale images, `X` is an M-by-N array. For single-frame true-color images, `X` is an M-by-N-by-3 array. Multiframe images are always 4-D arrays.

`X = dicomread(info)` reads the image data from the message referenced in the DICOM metadata structure `info`. The `info` structure is produced by the `dicominfo` function.

`[X,map] = dicomread(...)` returns the image `X` and the colormap `MAP`. If `X` is a grayscale or true-color image, `MAP` is empty.

`[X,map,alpha] = dicomread(...)` returns the image `X`, the colormap `map`, and an alpha channel matrix for `X`. The values of `alpha` are 0 if the pixel is opaque; otherwise they are row indices into `map`. The RGB value in `map` should be substituted for the value in `X` to use `alpha`. `alpha` has the same height and width as `X` and is 4-D for a multiframe image.

`[X,map,alpha,overlays] = dicomread(...)` also returns the image `X`, the colormap `map`, an alpha channel matrix for `X`, and any overlays from the DICOM file. Each overlay is a 1-bit black and white image with the same height and width as `X`. If multiple overlays are present in the file, `overlays` is a 4-D multiframe image. If no overlays are in the file, `overlays` is empty.

The first input argument, either `filename` or `info`, can be followed by a set of parameter name/value pairs.

```
[...] = dicomread(filename,param1, value1, param2, value2, ...)
[...] = dicomread(info, param1, value1, param2, value2, ...)
```

Supported parameters names and values include the following.

'Frames', V	dicomread reads only the frames in the vector V from the image. V must be an integer scalar, a vector of integers, or the string 'all'. The default value is 'all'.
'Dictionary', D	dicomread uses the data dictionary file whose filename is in the string D. The default value is 'dicom-dict.txt'.
'Raw', TF	dicomread performs pixel-level transformations depending on whether TF is 1 or 0. If TF is 1 (the default), dicomread reads the exact pixels from the image and no pixel-level transformations are performed. If TF is 0, images are rescaled to use the full dynamic range, and color images are automatically converted to the RGB colorspace. **Note 1:** Because the HSV colorspace is inadequately defined in the DICOM standard, dicomread does not automatically convert them to RGB. **Note 2:** dicomread never rescales or changes the color spaces of images containing signed data. **Note 3:** Rescaling values and applying colorspace conversions does not change the metadata in any way. Consequently, metadata values that refer to pixel values (such as window center/width or LUTs) may not be correct when pixels are scaled or converted.

Examples

Example 1

Use dicomread to retrieve the data matrix, X, and colormap matrix, map, needed to create a montage.

```
[X, map] = dicomread('US-PAL-8-10x-echo.dcm');
montage(X, map);
```

Example 2

Call dicomread with the information retrieved from the DICOM file using dicominfo. Display the image with imshow.

```
info = dicominfo('CT-MONO2-16-ankle.dcm');
Y = dicomread(info);
imshow(Y, []);
```

See Also

dicominfo, dicomwrite

Purpose

Write images as DICOM file

Syntax

```
dicomwrite(X, filename)
dicomwrite(X, map, filename)
dicomwrite(...,param1,value1,param2,value2,...)
dicomwrite(...,meta_struct,...)
dicomwrite(...,info,...)
status = dicomwrite(...)
```

Description

`dicomwrite(X,filename)` writes the binary, grayscale, or truecolor image, `X`, to the file, `filename`, where `filename` is a string specifying the name of the Digital Imaging and Communications in Medicine (DICOM) file to create.

`dicomwrite(X,map,filename)` writes the indexed image, `X`, with colormap, `map`.

`dicomwrite(...,param1,value1,param2,value2,...)` specifies additional metadata to write to the DICOM file. The parameters (`param1`, `param2`, etc.) are either names of DICOM file attributes or options that affect how the file is written. Each attribute or option has a corresponding value (`value1`, `value2`, etc.).

To find a list of the names of DICOM attributes, see the data dictionary file, `dicom-dict.txt`, included with the Image Processing Toolbox.

This table lists the options supported by the `dicomwrite` function.

Option Name	Description	Values
`'Endian'`	Specifies the byte-ordering for the file.	`'Little'` [Default] `'Big'`
`'VR'`	Specifies whether the two-letter value representation (VR) code should be written to the file (`'explicit'`) or inferred from the data dictionary (`'implicit'`).	`'Implicit'` [Default] `'Explicit'` Note: If you specify the `'Endian'` value `'Big'`, you can only specify the `'VR'` value of `'Explicit'`.

Option Name	Description	Values
'CompressionMode'	Specifies the type of compression to use when storing the image.	'None' [Default] 'JPEG lossy' 'RLE'
'TransferSyntax'	A DICOM UID specifying the DICOM Transfer Syntax. Note: If you specify the 'TransferSyntax' option, dicomwrite ignores the other three options, if they are specified. The 'TransferSyntax' option encodes the settings for the 'Endian', 'VR' and 'CompressionMode' options in a single value.	A DICOM Transfer Syntax that specifies the default values for 'Endian', 'VR', and 'CompressionMode' options.

dicomwrite(...,meta_struct,...) specifies metadata in a structure, meta_struct. The structure's field names must be the names of DICOM file attributes or options. The field's value is the value of that attribute or option.

dicomwrite(...,info,...) specifies metadata in the metadata structure, info, which is produced by the dicominfo function. For more information about this structure, see dicominfo.

status = dicomwrite(...) returns a structure that lists three types of metadata that were passed to dicomwrite:

- Metadata that does not affect how the DICOM file is written
- Metadata that does not pertain to the type of image being written
- Metadata that is not modifiable by a user

This syntax can be useful when you specify an info structure that was created by dicominfo to the dicomwrite function. An info structure can contain many fields. If no metadata was specified, dicomwrite returns an empty matrix ([]).

The structure returned by dicomwrite contains these three fields.

Field	Description
'dicominfo_fields'	A cell array containing the names of metadata passed to dicomwrite that does not affect how the file is written.
'wrong_IOD'	A cell array containing the names of attributes passed to dicomwrite that do not pertain to the type of image being written. (IOD=Information Object Definition)
'not_modifiable'	A cell array containing the names of valid metadata fields for the image that cannot be modified by the user.

Example

This example uses dicominfo to retrieve information about the contents of the sample DICOM file included with the Image Processing Toolbox. The example uses dicomread to read the data from the file and then writes the data into a new DICOM file, including the metadata from the original file.

```
info = dicominfo('CT-MONO2-16-ankle.dcm');
Y = dicomread(info);
status = dicomwrite(Y,'my_dicomfile.dcm',info);
status =

    dicominfo_fields: {12x1 cell}
           wrong_IOD: {21x1 cell}
      not_modifiable: {23x1 cell}

status.dicominfo_fields
ans =

    'BitDepth'
    'ColorType'
    'FileModDate'
    'FileSize'
    'FileStruct'
    'Filename'
```

```
'Format'
'FormatVersion'
'Height'
'SelectedFrames'
'StartOfPixelData'
'Width'
```

See Also `dicomread`, `dicominfo`

Purpose Perform dilation on a binary image

Note This function is obsolete and may be removed in future versions. Use `imdilate` instead.

Syntax

```
BW2 = dilate(BW1,SE)
BW2 = dilate(BW1,SE,alg)
BW2 = dilate(BW1,SE,...,n)
```

Description `BW2 = dilate(BW1,SE)` performs dilation on the binary image `BW1`, using the binary structuring element `SE`. `SE` is a matrix containing only 1's and 0's.

`BW2 = dilate(BW1,SE,`*`alg`*`)` performs dilation using the specified algorithm. *`alg`* is a string that can have one of these values:

- `'spatial'` (default) – processes the image in the spatial domain.
- `'frequency'` – processes the image in the frequency domain.

Both algorithms produce the same result, but they make different trade-offs between speed and memory use. The frequency algorithm is faster for large images and structuring elements than the spatial algorithm, but uses much more memory.

`BW2 = dilate(BW1,SE,...,n)` performs the dilation operation `n` times.

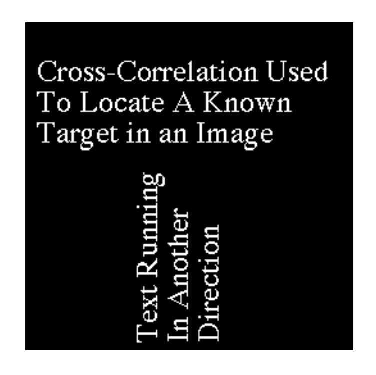
Class Support The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

Remarks You should use the frequency algorithm only if you have a large amount of memory on your system. If you use this algorithm with insufficient memory, it may actually be *slower* than the spatial algorithm, due to virtual memory paging. If the frequency algorithm slows down your system excessively, or if you receive "out of memory" messages, use the spatial algorithm instead.

Example

```
BW1 = imread('text.tif');
SE = ones(6,2);
BW2 = dilate(BW1,SE);
```

```
imshow(BW1)
figure, imshow(BW2)
```





See Also

`bwmorph`, `imdilate`, `imerode`

References

[1] Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992. p. 518.

[2] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992. p. 158.

Purpose

Convert an image, increasing apparent color resolution by dithering

Syntax

```
X = dither(RGB,map)
BW = dither(I)
```

Description

`X = dither(RGB,map)` creates an indexed image approximation of the RGB image in the array `RGB` by dithering the colors in colormap `map`. `map` can not have more than 65,536 colors.

`X = dither(RGB,map,Qm,Qe)` creates an indexed image from `RGB`, specifying the parameters `Qm` and `Qe`. `Qm` specifies the number of quantization bits to use along each color axis for the inverse color map, and `Qe` specifies the number of quantization bits to use for the color space error calculations. If `Qe < Qm`, dithering cannot be performed and an undithered indexed image is returned in `X`. If you omit these parameters, `dither` uses the default values `Qm` = 5, `Qe` = 8.

`BW = dither(I)` converts the intensity image in the matrix `I` to the binary (black and white) image `BW` by dithering.

Class Support

The input image, `RGB` or `I`, can be of class `uint8`, `uint16`, or `double`. All other input arguments must be of class `double`. The output indexed image, `X`, is of class `uint8` if it is an indexed image with 256 or fewer colors; otherwise its class is `uint16`. The output binary image, `BW`, is of class `logical`.

Algorithm

`dither` increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm.

References

[1] Floyd, R. W. and L. Steinberg. "An Adaptive Algorithm for Spatial Gray Scale," *International Symposium Digest of Technical Papers.* Society for Information Displays, 1975. p. 36.

[2] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 469-476.

See Also

`rgb2ind`

double

Purpose Convert data to double precision

`double` is a MATLAB built-in function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose Find edges in an intensity image

Syntax

```
BW = edge(I,'sobel')
BW = edge(I,'sobel',thresh)
BW = edge(I,'sobel',thresh,direction)
[BW,thresh] = edge(I,'sobel',...)

BW = edge(I,'prewitt')
BW = edge(I,'prewitt',thresh)
BW = edge(I,'prewitt',thresh,direction)
[BW,thresh] = edge(I,'prewitt',...)

BW = edge(I,'roberts')
BW = edge(I,'roberts',thresh)
[BW,thresh] = edge(I,'roberts',...)

BW = edge(I,'log')
BW = edge(I,'log',thresh)
BW = edge(I,'log',thresh,sigma)
[BW,threshold] = edge(I,'log',...)

BW = edge(I,'zerocross',thresh,h)
[BW,thresh] = edge(I,'zerocross',...)

BW = edge(I,'canny')
BW = edge(I,'canny',thresh)
BW = edge(I,'canny',thresh,sigma)
[BW,threshold] = edge(I,'canny',...)
```

Description `edge` takes an intensity image `I` as its input, and returns a binary image `BW` of the same size as `I`, with 1's where the function finds edges in `I` and 0's elsewhere.

`edge` supports six different edge-finding methods:

- The Sobel method finds edges using the Sobel approximation to the derivative. It returns edges at those points where the gradient of `I` is maximum.

- The Prewitt method finds edges using the Prewitt approximation to the derivative. It returns edges at those points where the gradient of `I` is maximum.
- The Roberts method finds edges using the Roberts approximation to the derivative. It returns edges at those points where the gradient of `I` is maximum.
- The Laplacian of Gaussian method finds edges by looking for zero crossings after filtering `I` with a Laplacian of Gaussian filter.
- The zero-cross method finds edges by looking for zero crossings after filtering `I` with a filter you specify.
- The Canny method finds edges by looking for local maxima of the gradient of `I`. The gradient is calculated using the derivative of a Gaussian filter. The method uses two thresholds, to detect strong and weak edges, and includes the weak edges in the output only if they are connected to strong edges. This method is therefore less likely than the others to be "fooled" by noise, and more likely to detect true weak edges.

The parameters you can supply differ depending on the method you specify. If you do not specify a method, `edge` uses the Sobel method.

Sobel Method

`BW = edge(I,'sobel')` specifies the Sobel method.

`BW = edge(I,'sobel',thresh)` specifies the sensitivity threshold for the Sobel method. `edge` ignores all edges that are not stronger than `thresh`. If you do not specify `thresh`, or if `thresh` is empty (`[]`), `edge` chooses the value automatically.

`BW = edge(I,'sobel',thresh,direction)` specifies direction of detection for the Sobel method. `direction` is a string specifying whether to look for `'horizontal'` or `'vertical'` edges, or `'both'` (the default).

`[BW,thresh] = edge(I,'sobel',...)` returns the threshold value.

Prewitt Method

`BW = edge(I,'prewitt')` specifies the Prewitt method.

`BW = edge(I,'prewitt',thresh)` specifies the sensitivity threshold for the Prewitt method. `edge` ignores all edges that are not stronger than `thresh`. If

you do not specify `thresh`, or if `thresh` is empty (`[]`), `edge` chooses the value automatically.

`BW = edge(I,'prewitt',thresh,direction)` specifies direction of detection for the Prewitt method. `direction` is a string specifying whether to look for `'horizontal'` or `'vertical'` edges, or `'both'` (the default).

`[BW,thresh] = edge(I,'prewitt',...)` returns the threshold value.

Roberts Method

`BW = edge(I,method)` specifies the Roberts method.

`BW = edge(I,method,thresh)` specifies the sensitivity threshold for the Roberts method. `edge` ignores all edges that are not stronger than `thresh`. If you do not specify `thresh`, or if `thresh` is empty (`[]`), `edge` chooses the value automatically.

`[BW,thresh] = edge(I,method,...)` returns the threshold value.

Laplacian of Gaussian Method

`BW = edge(I,'log')` specifies the Laplacian of Gaussian method.

`BW = edge(I,'log',thresh)` specifies the sensitivity threshold for the Laplacian of Gaussian method. `edge` ignores all edges that are not stronger than `thresh`. If you do not specify `thresh`, or if `thresh` is empty (`[]`), `edge` chooses the value automatically.

`BW = edge(I,'log',thresh,sigma)` specifies the Laplacian of Gaussian method, using `sigma` as the standard deviation of the LoG filter. The default `sigma` is 2; the size of the filter is `n`-by-`n`, where `n = ceil(sigma*3)*2+1`.

`[BW,thresh] = edge(I,'log',...)` returns the threshold value.

Zero-cross Method

`BW = edge(I,'zerocross',thresh,h)` specifies the zero-cross method, using the filter `h`. `thresh` is the sensitivity threshold; if the argument is empty (`[]`), `edge` chooses the sensitivity threshold automatically.

`[BW,thresh] = edge(I,'zerocross',...)` returns the threshold value.

Canny Method

BW = edge(I,'canny') specifies the Canny method.

BW = edge(I,'canny',thresh) specifies sensitivity thresholds for the Canny method. thresh is a two-element vector in which the first element is the low threshold, and the second element is the high threshold. If you specify a scalar for thresh, this value is used for the high threshold and 0.4*thresh is used for the low threshold. If you do not specify thresh, or if thresh is empty ([]), edge chooses low and high values automatically.

BW = edge(I,'canny',thresh,sigma) specifies the Canny method, using sigma as the standard deviation of the Gaussian filter. The default sigma is 1; the size of the filter is chosen automatically, based on sigma.

[BW,thresh] = edge(I,'canny',...) returns the threshold values as a two-element vector.

Class Support

I can be of class uint8, uint16, or double. BW is of class logical.

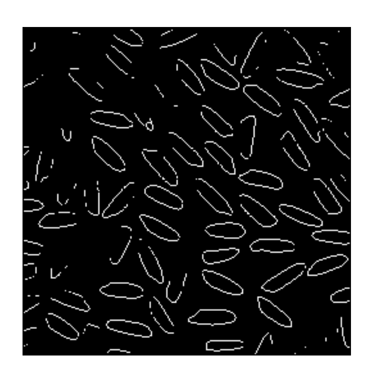
Remarks

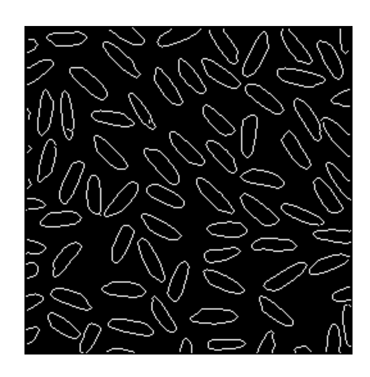
For the 'log' and 'zerocross' methods, if you specify a threshold of 0, the output image has closed contours, because it includes all of the zero crossings in the input image.

Example

Find the edges of the rice.tif image using the Prewitt and Canny methods.

```
I = imread('rice.tif');
BW1 = edge(I,'prewitt');
BW2 = edge(I,'canny');
imshow(BW1);
figure, imshow(BW2)
```





See Also

`fspecial`

References

[1] Canny, John. "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1986. Vol. PAMI-8, No. 6, pp. 679-698.

[2] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 478-488.

[3] Parker, James R. *Algorithms for Image Processing and Computer Vision.* New York: John Wiley & Sons, Inc., 1997. pp. 23-29.

edgetaper

Purpose Taper the discontinuities along the image edges

Syntax `J = edgetaper(I,PSF)`

Description `J = edgetaper(I,PSF)` blurs the edges of the input image, `I`, using the point spread function, `PSF`. The output image, `J`, is the weighted sum of the original image, `I`, and its blurred version. The weighting array, determined by the autocorrelation function of `PSF`, makes `J` equal to `I` in its central region, and equal to the blurred version of `I` near the edges.

The `edgetaper` function reduces the ringing effect in image deblurring methods that use the discrete Fourier transform, such as `deconvwnr`, `deconvreg`, and `deconvlucy`.

Note The size of the `PSF` cannot exceed half of the image size in any dimension.

Class Support `I` and `PSF` can be of class `uint8`, `uint16`, or `double`. `J` is of the same class as `I`.

Example

```
I = imread('cameraman.tif');
PSF = fspecial('gaussian',60,10);
J = edgetaper(I,PSF);
subplot(1,2,1);imshow(I,[]);title('original image');
subplot(1,2,2);imshow(J,[]);title('edges tapered');
```

See Also `deconvlucy`, `deconvreg`, `deconvwnr`, `otf2psf`, `padarray`, `psf2otf`

Purpose Perform erosion on a binary image

Note This function is obsolete and may be removed in future versions. Use `imerode` instead.

Syntax

```
BW2 = erode(BW1,SE)
BW2 = erode(BW1,SE,alg)
BW2 = erode(BW1,SE,...,n)
```

Description `BW2 = erode(BW1,SE)` performs erosion on the binary image `BW1`, using the binary structuring element `SE`. `SE` is a matrix containing only 1's and 0's.

`BW2 = erode(BW1,SE,alg)` performs erosion using the specified algorithm. *`alg`* is a string that can have one of these values:

- `'spatial'` (default) – processes the image in the spatial domain
- `'frequency'` – processes the image in the frequency domain

Both algorithms produce the same result, but they make different tradeoffs between speed and memory use. The frequency algorithm is faster for large images and structuring elements than the spatial algorithm, but uses much more memory.

`BW2 = erode(BW1,SE,...,n)` performs the erosion operation `n` times.

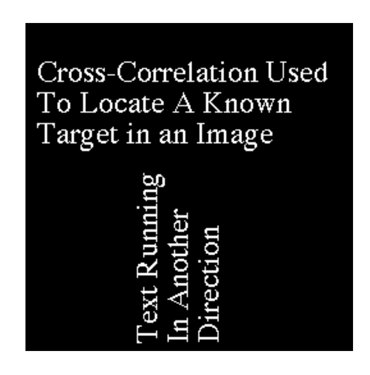
Class Support The input image `BW1` can be of class `double` or `uint8`. The output image `BW2` is of class `uint8`.

Remarks You should use the frequency algorithm only if you have a large amount of memory on your system. If you use this algorithm with insufficient memory, it may actually be *slower* than the spatial algorithm, due to virtual memory paging. If the frequency algorithm slows down your system excessively, or if you receive "out of memory" messages, use the spatial algorithm instead.

Example

```
BW1 = imread('text.tif');
SE = ones(3,1);
BW2 = erode(BW1,SE);
```

```
imshow(BW1)
figure, imshow(BW2)
```





See Also

bwmorph, imdilate, imerode

References

[1] Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992. p. 518.

[2] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I*. Addison-Wesley, 1992. p. 158.

Purpose Compute two-dimensional fast Fourier transform

`fft2` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose

Compute N-dimensional fast Fourier transform

`fftn` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose Shift zero-frequency component of fast Fourier transform to center of spectrum

`fftshift` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

filter2

Purpose Perform two-dimensional linear filtering

`filter2` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose Find output bounds for spatial transformation

Syntax

```
outbounds = findbounds(TFORM,inbounds)
```

Description

outbounds = findbounds(TFORM,inbounds) estimates the output bounds corresponding to a given spatial transformation and a set of input bounds. TFORM is a spatial transformation structure as returned by maketform. inbounds is 2-by-NUM_DIMS matrix. The first row of inbounds specifies the lower bounds for each dimension, and the second row specifies the upper bounds. NUM_DIMS has to be consistent with the ndims_in field of TFORM.

outbounds has the same form as inbounds. It is an estimate of the smallest rectangular region completely containing the transformed rectangle represented by the input bounds. Since outbounds is only an estimate, it may not completely contain the transformed input rectangle.

Notes

imtransform uses findbounds to compute the 'OutputBounds' parameter if the user does not provide it.

If TFORM contains a forward transformation (a nonempty forward_fcn field), then findbounds works by transforming the vertices of the input bounds rectangle and then taking minimum and maximum values of the result.

If TFORM does not contain a forward transformation, then findbounds estimates the output bounds using the Nelder-Mead optimization function fminsearch. If the optimization procedure fails, findbounds issues a warning and returns outbounds=inbounds.

Example

```
inbounds = [0 0; 1 1]
tform = maketform('affine',[2 0 0; .5 3 0; 0 0 1])
outbounds = findbounds(tform, inbounds)
```

See Also

cp2tform, imtransform, maketform, tformarray, tformfwd, tforminv

fliptform

Purpose Flip the input and output roles of a TFORM structure

Syntax TFLIP = fliptform(T)

Description TFLIP = fliptform(T) creates a new spatial transformation structure, a TFORM struct, by flipping the roles of the inputs and outputs in an existing TFORM struct.

Example

```
T = maketform('affine', [.5 0 0; .5 2 0; 0 0 1]);
T2 = fliptform(T)
```

The following are equivalent:

```
x = tformfwd([-3 7],T)
x = tforminv([-3 7],T2)
```

See Also maketform, tformfwd, tforminv

Purpose Determine frequency spacing for two-dimensional frequency response

`freqspace` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

freqz2

Purpose

Compute two-dimensional frequency response

Syntax

```
[H,f1,f2] = freqz2(h,n1,n2)
[H,f1,f2] = freqz2(h,[n2 n1])
[H,f1,f2] = freqz2(h,f1,f2)
[H,f1,f2] = freqz2(h)
[...] = freqz2(h,...,[dx dy])
[...] = freqz2(h,...,dx)
freqz2(...)
```

Description

`[H,f1,f2] = freqz2(h,n1,n2)` returns `H`, the `n2`-by-`n1` frequency response of `h`, and the frequency vectors `f1` (of length `n1`) and `f2` (of length `n2`). `h` is a two-dimensional FIR filter, in the form of a computational molecule. `f1` and `f2` are returned as normalized frequencies in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians.

`[H,f1,f2] = freqz2(h,[n2 n1])` returns the same result returned by `[H,f1,f2] = freqz2(h,n1,n2)`.

`[H,f1,f2] = freqz2(h)` uses `[n2 n1] = [64 64]`.

`[H,f1,f2] = freqz2(h,f1,f2)` returns the frequency response for the FIR filter `h` at frequency values in `f1` and `f2`. These frequency values must be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians.

`[...] = freqz2(h,...,[dx dy])` uses `[dx dy]` to override the intersample spacing in `h`. `dx` determines the spacing for the x-dimension and `dy` determines the spacing for the y-dimension. The default spacing is 0.5, which corresponds to a sampling frequency of 2.0.

`[...] = freqz2(h,...,dx)` uses `dx` to determine the intersample spacing in both dimensions.

With no output arguments, `freqz2(...)` produces a mesh plot of the two-dimensional magnitude frequency response.

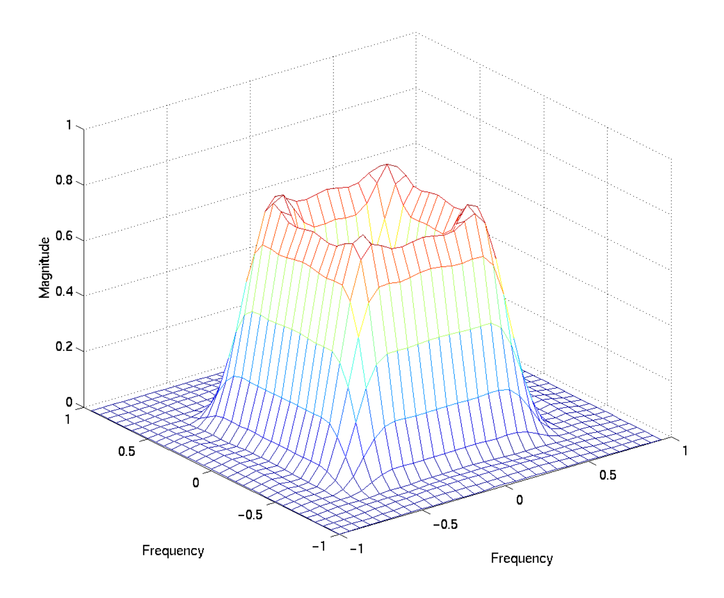
Class Support

The input matrix `h` can be of class `double` or of any integer class. All other inputs to `freqz2` must be of class `double`. All outputs are of class `double`.

Example

Use the window method to create a 16-by-16 filter, then view its frequency response using `freqz2`.

```
Hd = zeros(16,16);
Hd(5:12,5:12) = 1;
Hd(7:10,7:10) = 0;
h = fwind1(Hd,bartlett(16));
colormap(jet(64))
freqz2(h,[32 32]); axis ([-1 1 -1 1 0 1])
```



See Also

`freqz` in the *Signal Processing Toolbox User's Guide*

fsamp2

Purpose

Design two-dimensional FIR filter using frequency sampling

Syntax

```
h = fsamp2(Hd)
h = fsamp2(f1,f2,Hd,[m n])
```

Description

`fsamp2` designs two-dimensional FIR filters based on a desired two-dimensional frequency response sampled at points on the Cartesian plane.

`h = fsamp2(Hd)` designs a two-dimensional FIR filter with frequency response `Hd`, and returns the filter coefficients in matrix `h`. (`fsamp2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`.) The filter `h` has a frequency response that passes through points in `Hd`. If `Hd` is `m`-by-`n`, then `h` is also `m`-by-`n`.

`Hd` is a matrix containing the desired frequency response sampled at equally spaced points between -1.0 and 1.0 along the *x* and *y* frequency axes, where 1.0 corresponds to half the sampling frequency, or π radians.

$$H_d(f_1, f_2) = H_d(\omega_1, \omega_2)\Big|_{\omega_1 = \pi f_1,\ \omega_2 = \pi f_2}$$

For accurate results, use frequency points returned by `freqspace` to create `Hd`. (See the entry for `freqspace` for more information.)

`h = fsamp2(f1,f2,Hd,[m n])` produces an `m`-by-`n` FIR filter by matching the filter response at the points in the vectors `f1` and `f2`. The frequency vectors `f1` and `f2` are in normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians. The resulting filter fits the desired response as closely as possible in the least squares sense. For best results, there must be at least `m*n` desired frequency points. `fsamp2` issues a warning if you specify fewer than `m*n` points.

Class Support

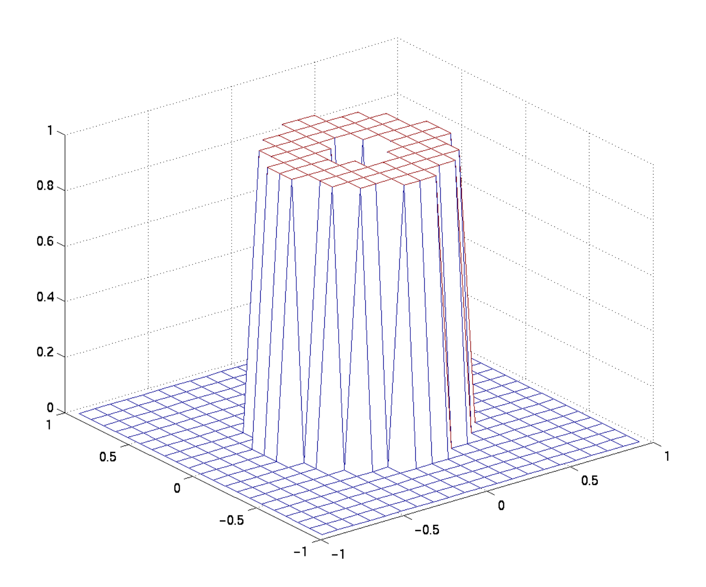
The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fsamp2` must be of class `double`. All outputs are of class `double`.

Example

Use `fsamp2` to design an approximately symmetric two-dimensional bandpass filter with passband between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians):

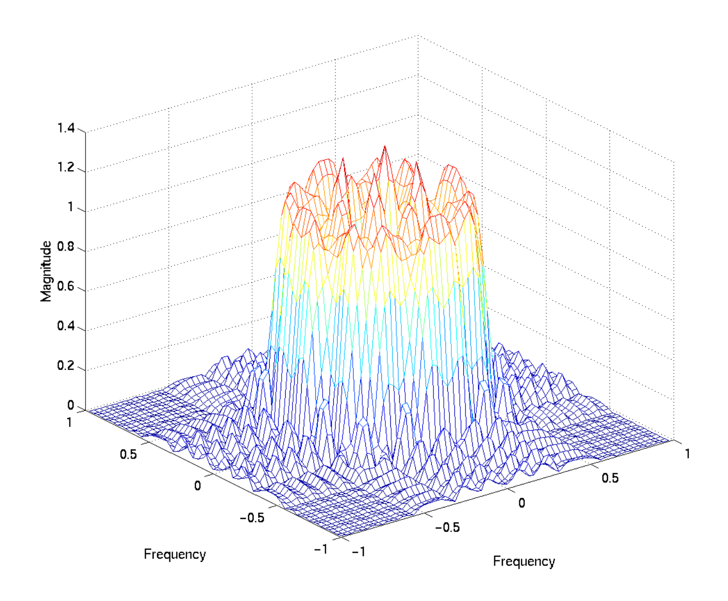
1. Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors `f1` and `f2`.

```
[f1,f2] = freqspace(21,'meshgrid');
Hd = ones(21);
r = sqrt(f1.^2 + f2.^2);
Hd((r<0.1)|(r>0.5)) = 0;
colormap(jet(64))
mesh(f1,f2,Hd)
```



2 Design the filter that passes through this response.

```
h = fsamp2(Hd);
freqz2(h)
```



Algorithm

`fsamp2` computes the filter `h` by taking the inverse discrete Fourier transform of the desired frequency response. If the desired frequency response is real and symmetric (zero phase), the resulting filter is also zero phase.

See Also

`conv2`, `filter2`, `freqspace`, `ftrans2`, `fwind1`, `fwind2`

Reference

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 213-217.

Purpose

Create 2-D special filters

Syntax

```
h = fspecial(type)
h = fspecial(type,parameters)
```

Description

`h = fspecial(type)` creates a two-dimensional filter, `h`, of the specified *type*. `fspecial` returns `h` as a correlation kernel, which is the appropriate form to use with `imfilter`. *type* is a string having one of these values:

- `'gaussian'` for a Gaussian lowpass filter
- `'sobel'` for a Sobel horizontal edge-emphasizing filter
- `'prewitt'` for a Prewitt horizontal edge-emphasizing filter
- `'laplacian'` for a filter approximating the two-dimensional Laplacian operator
- `'log'` for a Laplacian of Gaussian filter
- `'average'` for an averaging filter
- `'unsharp'` for an unsharp contrast enhancement filter

`h = fspecial(type,parameters)` accepts a filter *type* plus additional modifying `parameters` particular to the type of filter chosen. If you omit these arguments, `fspecial` uses default values for the `parameters`.

The following list shows the syntax for each filter type. Where applicable, additional parameters are also shown.

- `h = fspecial('average',hsize)` returns an averaging filter, `h`, of size `hsize`. The argument `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[3 3]`.
- `h = fspecial('disk',radius)` returns a circular averaging filter (pillbox) within the square matrix of side `2*radius+1`. The default `radius` is 5.
- `h = fspecial('gaussian',hsize,sigma)` returns a rotationally symmetric Gaussian lowpass filter of size `hsize` with standard deviation `sigma` (positive). `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[3 3]`; the default value for `sigma` is 0.5.
- `h = fspecial('laplacian',alpha)` returns a 3-by-3 filter approximating the shape of the two-dimensional Laplacian operator. The parameter `alpha`

controls the shape of the Laplacian and must be in the range 0.0 to 1.0. The default value for `alpha` is 0.2.

- `h = fspecial('log',hsize,sigma)` returns a rotationally symmetric Laplacian of Gaussian filter of size `hsize` with standard deviation `sigma` (positive). `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[5 5]` and 0.5 for `sigma`.
- `h = fspecial('motion',len,theta)` returns a filter to approximate, once convolved with an image, the linear motion of a camera by `len` pixels, with an angle of `theta` degrees in a counter-clockwise direction. The filter becomes a vector for horizontal and vertical motions. The default `len` is 9, the default `theta` is 0, which corresponds to a horizontal motion of 9 pixels.
- `h = fspecial('prewitt')` returns a 3-by-3 filter, `h`, (shown below) that emphasizes horizontal edges by approximating a vertical gradient. If you need to emphasize vertical edges, transpose the filter, `h'`.

  ```
  [ 1  1  1
    0  0  0
   -1 -1 -1 ]
  ```

  To find vertical edges, or for *x*-derivatives, use `h'`.
- `h = fspecial('sobel')` returns a 3-by-3 filter, `h`, (shown below) that emphasizes horizontal edges using the smoothing effect by approximating a vertical gradient. If you need to emphasize vertical edges, transpose the filter, `h'`.

  ```
  [ 1  2  1
    0  0  0
   -1 -2 -1 ]
  ```
- `h = fspecial('unsharp',alpha)` returns a 3-by-3 unsharp contrast enhancement filter. `fspecial` creates the unsharp filter from the negative of the Laplacian filter with parameter `alpha`. `alpha` controls the shape of the Laplacian and must be in the range 0.0 to 1.0. The default value for `alpha` is 0.2.

Class Support

`h` is of class double.

Example

```
I = imread('saturn.tif');
subplot(2,2,1);imshow(I);title('Original Image');
```

```
H = fspecial('motion',50,45);
MotionBlur = imfilter(I,H);
subplot(2,2,2);imshow(MotionBlur);title('Motion Blurred Image');
H = fspecial('disk',10);
blurred = imfilter(I,H);
subplot(2,2,3);imshow(blurred);title('Blurred Image');
H = fspecial('unsharp');
sharpened = imfilter(I,H);
subplot(2,2,4);imshow(sharpened);title('Sharpened Image');
```

Original Image

Motion Blurred Image

Blurred Image

Sharpened Image

Algorithms

`fspecial` creates Gaussian filters using

$$h_g(n_1, n_2) = e^{-(n_1^2 + n_2^2)/(2\sigma^2)}$$

$$h(n_1, n_2) = \frac{h_g(n_1, n_2)}{\sum_{n_1}\sum_{n_2} h_g}$$

`fspecial` creates Laplacian filters using

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

$$\nabla^2 \approx \frac{4}{(\alpha + 1)}\begin{bmatrix} \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \\ \frac{1-\alpha}{4} & -1 & \frac{1-\alpha}{4} \\ \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \end{bmatrix}$$

`fspecial` creates Laplacian of Gaussian (LoG) filters using

$$h_g(n_1, n_2) = e^{-(n_1^2 + n_2^2)/(2\sigma^2)}$$

$$h(n_1, n_2) = \frac{({n_1}^2 + {n_2}^2 - 2\sigma^2) h_g(n_1, n_2)}{2\pi\sigma^6 \sum_{n_1}\sum_{n_2} h_g}$$

`fspecial` creates averaging filters using

```
ones(n(1),n(2))/(n(1)*n(2))
```

`fspecial` creates unsharp filters using

$$\frac{1}{(\alpha + 1)}\begin{bmatrix} -\alpha & \alpha - 1 & -\alpha \\ \alpha - 1 & \alpha + 5 & \alpha - 1 \\ -\alpha & \alpha - 1 & -\alpha \end{bmatrix}$$

See Also

`conv2`, `edge`, `filter2`, `fsamp2`, `fwind1`, `fwind2`, `imfilter`

`del2` in the MATLAB Function Reference

ftrans2

Purpose

Design two-dimensional FIR filter using frequency transformation

Syntax

```
h = ftrans2(b,t)
h = ftrans2(b)
```

Description

`h = ftrans2(b,t)` produces the two-dimensional FIR filter `h` that corresponds to the one-dimensional FIR filter `b` using the transform `t`. (`ftrans2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`.) `b` must be a one-dimensional, odd-length (Type I) FIR filter such as can be returned by `fir1`, `fir2`, or `remez` in the Signal Processing Toolbox. The transform matrix `t` contains coefficients that define the frequency transformation to use. If `t` is `m`-by-`n` and `b` has length `Q`, then `h` is size `((m-1)*(Q-1)/2+1)`-by-`((n-1)*(Q-1)/2+1)`.

`h = ftrans2(b)` uses the McClellan transform matrix `t`.

```
t = [1 2 1; 2 -4 2; 1 2 1]/8;
```

Remarks

The transformation below defines the frequency response of the two-dimensional filter returned by `ftrans2`,

$$H(\omega_1, \omega_2) = B(\omega)\big|_{\cos\omega = T(\omega_1, \omega_2)}$$

where $B(\omega)$ is the Fourier transform of the one-dimensional filter `b`,

$$B(\omega) = \sum_{n=-N}^{N} b(n)e^{-j\omega n}$$

and $T(\omega_1,\omega_2)$ is the Fourier transform of the transformation matrix `t`.

$$T(\omega_1, \omega_2) = \sum_{n_2}\sum_{n_1} t(n_1, n_2)e^{-j\omega_1 n_1}e^{-j\omega_2 n_2}$$

The returned filter `h` is the inverse Fourier transform of $H(\omega_1,\omega_2)$.

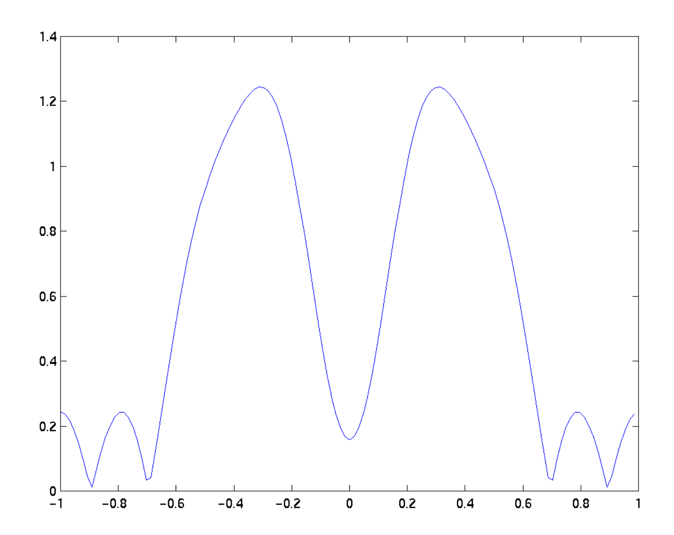
$$h(n_1, n_2) = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} H(\omega_1, \omega_2)e^{j\omega_1 n_1}e^{j\omega_2 n_2}d\omega_1 d\omega_2$$

Example

Use `ftrans2` to design an approximately circularly symmetric two-dimensional bandpass filter with passband between 0.1 and 0.6 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians):

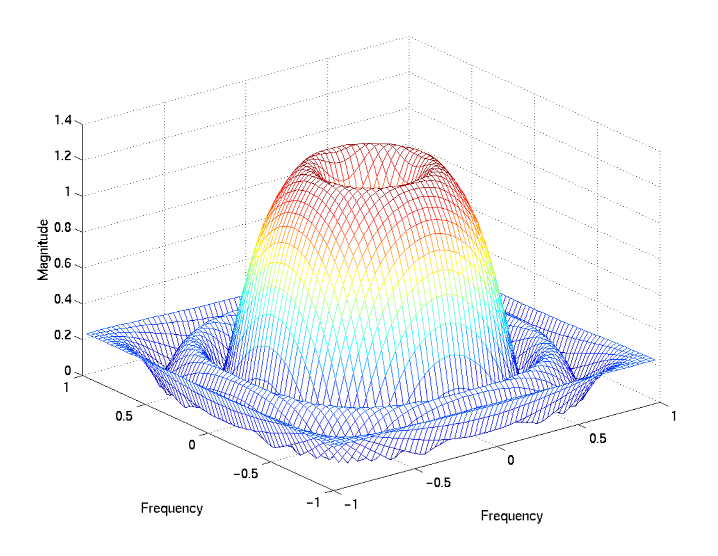
1 Since `ftrans2` transforms a one-dimensional FIR filter to create a two-dimensional filter, first design a one-dimensional FIR bandpass filter using the Signal Processing Toolbox function `remez`.

```
colormap(jet(64))
b = remez(10,[0 0.05 0.15 0.55 0.65 1],[0 0 1 1 0 0]);
[H,w] = freqz(b,1,128,'whole');
plot(w/pi-1,fftshift(abs(H)))
```



2 Use `ftrans2` with the default McClellan transformation to create the desired approximately circularly symmetric filter.

```
h = ftrans2(b);
freqz2(h)
```



See Also `conv2`, `filter2`, `fsamp2`, `fwind1`, `fwind2`

Reference

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 218-237.

Purpose Design two-dimensional FIR filter using one-dimensional window method

Syntax

```
h = fwind1(Hd,win)
h = fwind1(Hd,win1,win2)
h = fwind1(f1,f2,Hd,...)
```

Description

`fwind1` designs two-dimensional FIR filters using the window method. `fwind1` uses a one-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response `Hd`. `fwind1` works with one-dimensional windows only; use `fwind2` to work with two-dimensional windows.

`h = fwind1(Hd,win)` designs a two-dimensional FIR filter `h` with frequency response `Hd`. (`fwind1` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`.) `fwind1` uses the one-dimensional window `win` to form an approximately circularly symmetric two-dimensional window using Huang's method. You can specify `win` using windows from the Signal Processing Toolbox, such as `boxcar`, `hamming`, `hanning`, `bartlett`, `blackman`, `kaiser`, or `chebwin`. If `length(win)` is `n`, then `h` is `n`-by-`n`.

`Hd` is a matrix containing the desired frequency response sampled at equally spaced points between -1.0 and 1.0 (in normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians) along the *x* and *y* frequency axes. For accurate results, use frequency points returned by `freqspace` to create `Hd`. (See the entry for `freqspace` for more information.)

`h = fwind1(Hd,win1,win2)` uses the two one-dimensional windows `win1` and `win2` to create a separable two-dimensional window. If `length(win1)` is `n` and `length(win2)` is `m`, then `h` is `m`-by-`n`.

`h = fwind1(f1,f2,Hd,...)` lets you specify the desired frequency response `Hd` at arbitrary frequencies (`f1` and `f2`) along the *x* and *y* axes. The frequency vectors `f1` and `f2` should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians. The length of the window(s) controls the size of the resulting filter, as above.

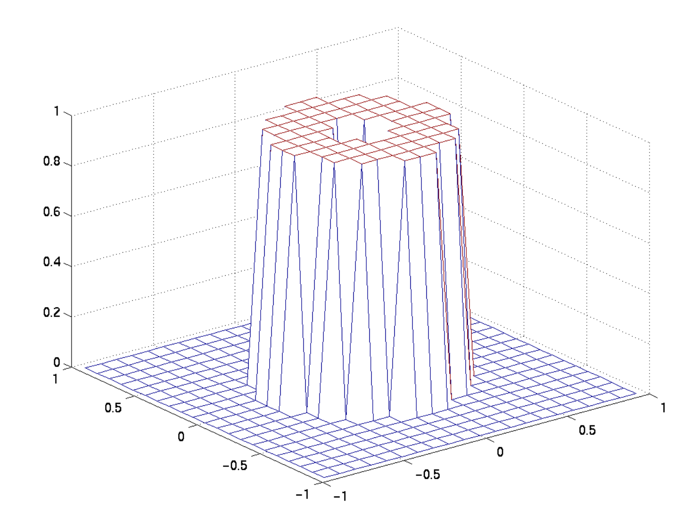
Class Support The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind1` must be of class `double`. All outputs are of class `double`.

Example

Use `fwind1` to design an approximately circularly symmetric two-dimensional bandpass filter with passband between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians):

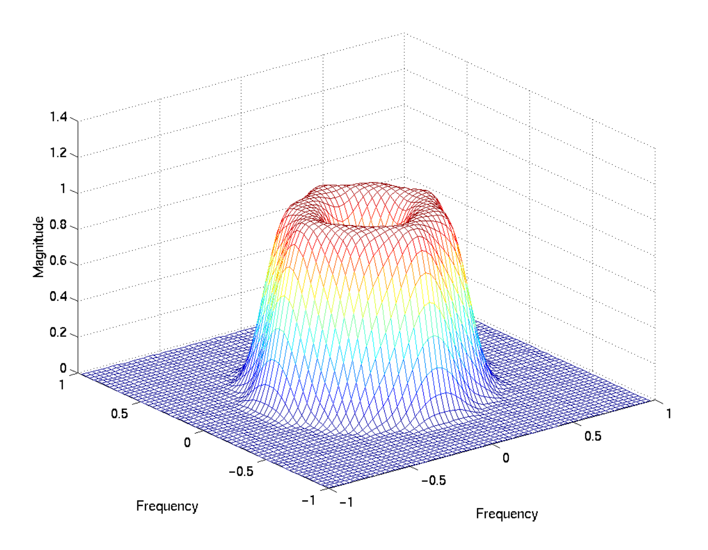
1 Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors `f1` and `f2`.

```
[f1,f2] = freqspace(21,'meshgrid');
Hd = ones(21);
r = sqrt(f1.^2 + f2.^2);
Hd((r<0.1)|(r>0.5)) = 0;
colormap(jet(64))
mesh(f1,f2,Hd)
```



2 Design the filter using a one-dimensional Hamming window.

```
h = fwind1(Hd,hamming(21));
freqz2(h)
```



Algorithm

`fwind1` takes a one-dimensional window specification and forms an approximately circularly symmetric two-dimensional window using Huang's method,

$$w(n_1, n_2) = w(t)\big|_{t = \sqrt{n_1^2 + n_2^2}}$$

where $w(t)$ is the one-dimensional window and $w(n_1,n_2)$ is the resulting two-dimensional window.

Given two windows, `fwind1` forms a separable two-dimensional window.

$$w(n_1, n_2) = w_1(n_1)w_2(n_2)$$

`fwind1` calls `fwind2` with `Hd` and the two-dimensional window. `fwind2` computes `h` using an inverse Fourier transform and multiplication by the two-dimensional window.

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} H_d(\omega_1, \omega_2)e^{j\omega_1 n_1}e^{j\omega_2 n_2}d\omega_1 d\omega_2$$

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_1, n_2)$$

See Also `conv2`, `filter2`, `fsamp2`, `freqspace`, `ftrans2`, `fwind2`

Reference

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990.

Purpose

Design two-dimensional FIR filter using two-dimensional window method

Syntax

```
h = fwind2(Hd,win)
h = fwind2(f1,f2,Hd,win)
```

Description

Use `fwind2` to design two-dimensional FIR filters using the window method. `fwind2` uses a two-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response `Hd`. `fwind2` works with two-dimensional windows; use `fwind1` to work with one-dimensional windows.

`h = fwind2(Hd,win)` produces the two-dimensional FIR filter `h` using an inverse Fourier transform of the desired frequency response `Hd` and multiplication by the window `win`. `Hd` is a matrix containing the desired frequency response at equally spaced points in the Cartesian plane. `fwind2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`. `h` is the same size as `win`.

For accurate results, use frequency points returned by `freqspace` to create `Hd`. (See the entry for `freqspace` for more information.

`h = fwind2(f1,f2,Hd,win)` lets you specify the desired frequency response `Hd` at arbitrary frequencies (`f1` and `f2`) along the *x* and *y* axes. The frequency vectors `f1` and `f2` should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians. `h` is the same size as `win`.

Class Support

The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind2` must be of class `double`. All outputs are of class `double`.

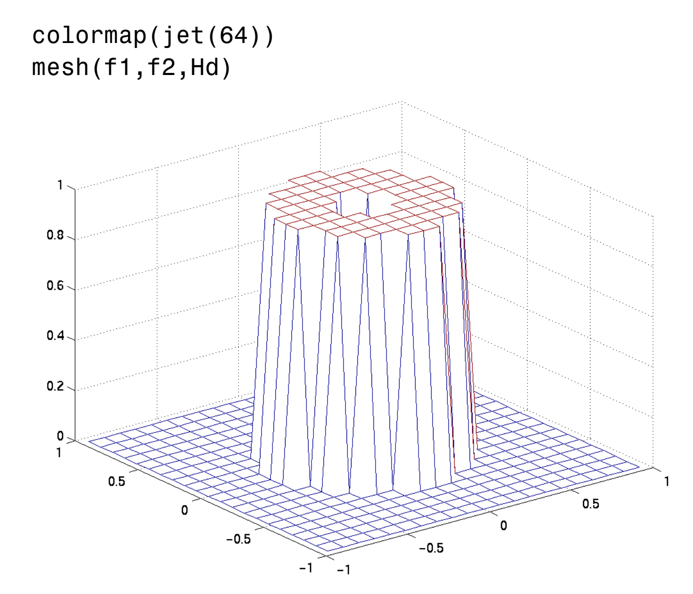
Example

Use `fwind2` to design an approximately circularly symmetric two-dimensional bandpass filter with passband between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians):

1 Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency range vectors `f1` and `f2`.

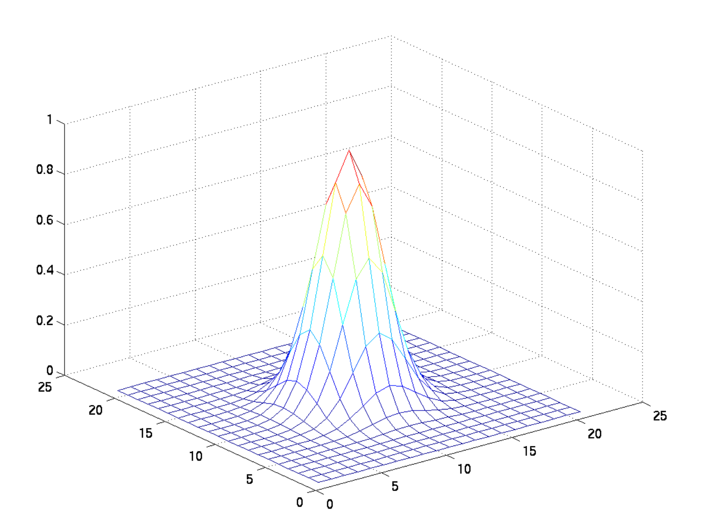
```
[f1,f2] = freqspace(21,'meshgrid');
Hd = ones(21);
r = sqrt(f1.^2 + f2.^2);
Hd((r<0.1)|(r>0.5)) = 0;
```

```
colormap(jet(64))
mesh(f1,f2,Hd)
```



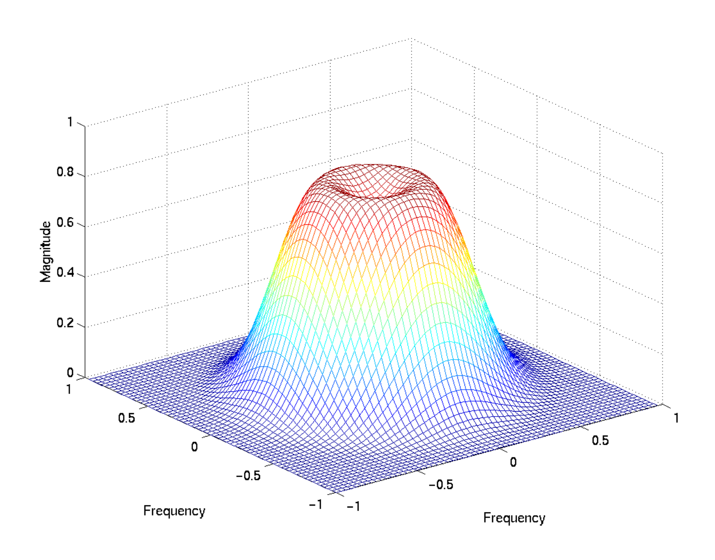
2 Create a two-dimensional Gaussian window using `fspecial`.

```
win = fspecial('gaussian',21,2);
win = win ./ max(win(:));  % Make the maximum window value be 1.
mesh(win)
```



3 Design the filter using the window from step 2.

```
h = fwind2(Hd,win);
freqz2(h)
```



Algorithm

fwind2 computes h using an inverse Fourier transform and multiplication by the two-dimensional window win.

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} H_d(\omega_1, \omega_2)e^{j\omega_1 n_1}e^{j\omega_2 n_2}d\omega_1 d\omega_2$$

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_1, n_2)$$

See Also

conv2, filter2, fsamp2, freqspace, ftrans2, fwind1

Reference

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 202-213.

Purpose Get height of structuring element

Syntax

```
H = getheight(SE)
```

Description `H = getheight(SE)` returns an array the same size as `getnhood(SE)` containing the height associated with each of the structuring element neighbors. `H` is all zeros for a flat structuring element.

Class Support `SE` is a `STREL` object. `H` is of class `double`.

Example

```
se = strel(ones(3,3),magic(3));
getheight(se)
```

See Also `strel`, `getnhood`

getimage

Purpose Get image data from axes

Syntax

```
A = getimage(h)
[x,y,A] = getimage(h)
[...,A,flag] = getimage(h)
[...] = getimage
```

Description

`A = getimage(h)` returns the first image data contained in the Handle Graphics object `h`. `h` can be a figure, axes, image, or texture-mapped surface. `A` is identical to the image `CData`; it contains the same values and is of the same class (`uint8` or `double`) as the image `CData`. If `h` is not an image or does not contain an image or texture-mapped surface, `A` is empty.

`[x,y,A] = getimage(h)` returns the image `XData` in `x` and the `YData` in `y`. `XData` and `YData` are two-element vectors that indicate the range of the x-axis and y-axis.

`[...,A,flag] = getimage(h)` returns an integer flag that indicates the type of image `h` contains. This table summarizes the possible values for `flag`.

Flag	Type of Image
0	Not an image; `A` is returned as an empty matrix
1	Intensity image with values in standard range ([0,1] for `double` arrays, [0,255] for `uint8` arrays, [0,65535] for `uint16` arrays)
2	Indexed image
3	Intensity data, but not in standard range
4	RGB image

`[...] = getimage` returns information for the current axes. It is equivalent to `[...] = getimage(gca)`.

Class Support

The output array `A` is of the same class as the image `CData`. All other inputs and outputs are of class `double`.

Example

This example illustrates obtaining the image data from an image displayed directly from a file.

```
imshow rice.tif
I = getimage;
```

getneighbors

Purpose Get structuring element neighbor locations and heights

Syntax

```
[offsets,heights] = getneighbors(SE)
```

Description

`[offsets,heights] = getneighbors(SE)` returns the relative locations and corresponding heights for each of the neighbors in the structuring element object `SE`.

`offsets` is a P-by-N array where P is the number of neighbors in the structuring element and N is the dimensionality of the structuring element. Each row of `offsets` contains the location of the corresponding neighbor, relative to the center of the structuring element.

`heights` is a P-element column vector containing the height of each structuring element neighbor.

Class Support

`SE` is a `STREL` object. The return values, `offsets` and `heights`, are arrays of double precision values.

Example

```
se = strel([1 0 1],[5 0 -5])
[offsets,heights] = getneighbors(se)
se =
Nonflat STREL object containing 2 neighbors.

Neighborhood:
     1     0     1

Height:
     5     0    -5

offsets =
     0    -1
     0     1
heights =
     5    -5
```

See Also

`strel`, `getnhood`, `getheight`

Purpose Get structuring element neighborhood

Syntax

```
nhood = getnhood(SE)
```

Description `nhood = getnhood(SE)` returns the neighborhood associated with the structuring element `SE`.

Class Support `SE` is a `STREL` object. `nhood` is a `logical` array.

Example

```
se = strel(eye(5));
nhood = getnhood(se)
```

See Also `strel`, `getneighbors`

getsequence

Purpose Extract sequence of decomposed structuring elements

Syntax `SEQ = getsequence(SE)`

Description `SEQ = getsequence(SE)`, where `SE` is a structuring element array, returns another structuring element array `SEQ` containing the individual structuring elements that form the decomposition of `SE`. `SEQ` is equivalent to `SE`, but the elements of `SEQ` have no decomposition.

Class Support `SE` and `SEQ` are arrays of `STREL` objects.

Example The `strel` function uses decomposition for square structuring elements larger than 3-by-3. Use `getsequence` to extract the decomposed structuring elements.

```
se = strel('square',5)
seq = getsequence(se)
se =
Flat STREL object containing 25 neighbors.
Decomposition: 2 STREL objects containing a total of 10 neighbors

Neighborhood:
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
seq =
2x1 array of STREL objects
```

Use `imdilate` with the `'full'` option to see that dilating sequentially with the decomposed structuring elements really does form a 5-by-5 square:

```
imdilate(1,seq,'full')
```

See Also `imdilate`, `imerode`, `strel`

Purpose Convert an intensity image to an indexed image

Syntax

```
[X,map] = gray2ind(I,n)
[X,map] = gray2ind(BW,n)
```

Description `gray2ind` scales, then rounds, an intensity image to produce an equivalent indexed image.

`[X,map] = gray2ind(I,n)` converts the intensity image `I` to an indexed image `X` with colormap `gray(n)`. If `n` is omitted, it defaults to 64.

`[X,map] = gray2ind(BW,n)` converts the binary image, `BW`, to an indexed image, `X`, with colormap `gray(n)`. If `n` is omitted, it defaults to 2.

`n` must be an integer between 1 and 65536.

Class Support The input image, `I`, must be a real, nonsparse array of class `logical`, `uint8`, `uint16`, or `double`. It can have any dimension. The class of the output image, `X`, is `uint8` if the colormap length is less than or equal to 256; otherwise it is `uint16`.

See Also `ind2gray`

grayslice

Purpose Create indexed image from intensity image using multilevel thresholding

Syntax

```
X = grayslice(I,n)
X = grayslice(I,v)
```

Description

`X = grayslice(I,n)` thresholds the intensity image `I` using cutoff values $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$, returning an indexed image in `X`.

`X = grayslice(I,v)`, where `v` is a vector of values between 0 and 1, thresholds `I` using the values of `v`, returning an indexed image in `X`.

You can view the thresholded image using `imshow(X,map)` with a colormap of appropriate length.

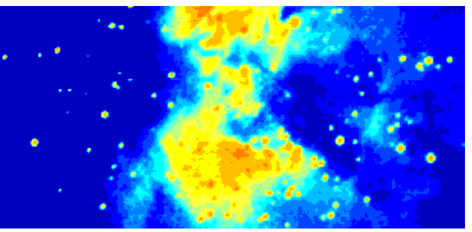
Class Support

The input image, `I`, can be of class `uint8`, `uint16`, or `double`. Note that the threshold values are always between 0 and 1, even if `I` is of class `uint8` or `uint16`. In this case, each threshold value is multiplied by 255 or 65535 to determine the actual threshold to use.

The class of the output image, `X`, depends on the number of threshold values, as specified by `n` or `length(v)`. If the number of threshold values is less than 256, then `X` is of class `uint8`, and the values in `X` range from 0 to `n` or `length(v)`. If the number of threshold values is 256 or greater, `X` is of class `double`, and the values in `X` range from 1 to `n+1` or `length(v)+1`.

Example

```
I = imread('ngc4024m.tif');
X = grayslice(I,16);
imshow(I)
figure, imshow(X,jet(16))
```

See Also `gray2ind`

Purpose

Compute global image threshold using Otsu's method

Syntax

```
level = graythresh(I)
```

Description

`level = graythresh(I)` computes a global threshold (`level`) that can be used to convert an intensity image to a binary image with `im2bw`.

`level` is a normalized intensity value that lies in the range `[0, 1]`.

The `graythresh` function uses Otsu's method, which chooses the threshold to minimize the intraclass variance of the black and white pixels.

Multidimensional arrays are converted automatically to 2-D arrays using `reshape`. The `graythresh` function ignores any nonzero imaginary part of `I`.

Class Support

The input image, `I`, can be of class `uint8`, `uint16`, or `double` and it must be nonsparse. The return value, `level`, is a double scalar.

See Also

`im2bw`

Reference

N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62-66, 1979.

histeq

Purpose Enhance contrast using histogram equalization

Syntax

```
J = histeq(I,hgram)
J = histeq(I,n)
[J,T] = histeq(I,...)

newmap = histeq(X,map,hgram)
newmap = histeq(X,map)
[newmap,T] = histeq(X,...)
```

Description

`histeq` enhances the contrast of images by transforming the values in an intensity image, or the values in the colormap of an indexed image, so that the histogram of the output image approximately matches a specified histogram.

`J = histeq(I,hgram)` transforms the intensity image `I` so that the histogram of the output intensity image `J` with `length(hgram)` bins approximately matches `hgram`. The vector `hgram` should contain integer counts for equally spaced bins with intensity values in the appropriate range: [0, 1] for images of class `double`, [0, 255] for images of class `uint8`, and [0, 65535] for images of class `uint16`. `histeq` automatically scales `hgram` so that `sum(hgram) = prod(size(I))`. The histogram of `J` will better match `hgram` when `length(hgram)` is much smaller than the number of discrete levels in `I`.

`J = histeq(I,n)` transforms the intensity image `I`, returning in `J` an intensity image with `n` discrete gray levels. A roughly equal number of pixels is mapped to each of the `n` levels in `J`, so that the histogram of `J` is approximately flat. (The histogram of `J` is flatter when `n` is much smaller than the number of discrete levels in `I`.) The default value for `n` is 64.

`[J,T] = histeq(I,...)` returns the grayscale transformation that maps gray levels in the intensity image `I` to gray levels in `J`.

`newmap = histeq(X,map,hgram)` transforms the colormap associated with the indexed image `X` so that the histogram of the gray component of the indexed image `(X,newmap)` approximately matches `hgram`. The `histeq` function returns the transformed colormap in `newmap`. `length(hgram)` must be the same as `size(map,1)`.

`newmap = histeq(X,map)` transforms the values in the colormap so that the histogram of the gray component of the indexed image `X` is approximately flat. It returns the transformed colormap in `newmap`.

`[newmap,T] = histeq(X,...)` returns the grayscale transformation `T` that maps the gray component of `map` to the gray component of `newmap`.

Class Support

For syntaxes that include an intensity image `I` as input, `I` can be of class `uint8`, `uint16`, or `double`, and the output image `J` has the same class as `I`. For syntaxes that include an indexed image `X` as input, `X` can be of class `uint8` or `double`; the output colormap is always of class `double`. Also, the optional output `T` (the gray level transform) is always of class `double`.

Example

Enhance the contrast of an intensity image using histogram equalization.

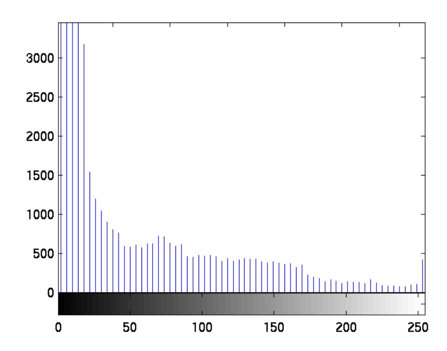
```
I = imread('tire.tif');
J = histeq(I);
imshow(I)
figure, imshow(J)
```

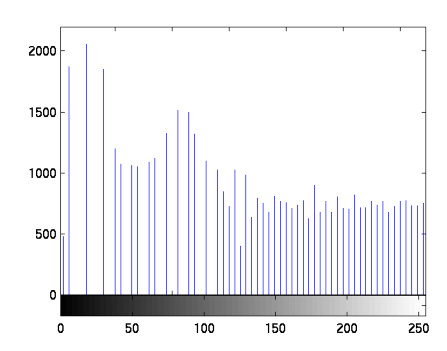




Display the resulting histograms.

```
imhist(I,64)
figure; imhist(J,64)
```





Algorithm

When you supply a desired histogram `hgram`, `histeq` chooses the grayscale transformation T to minimize

$$\left|c_1(T(k)) - c_0(k)\right|$$

where c_0 is the cumulative histogram of `A`, c_1 is the cumulative sum of `hgram` for all intensities k. This minimization is subject to the constraints that T must be monotonic and $c_1(T(a))$ cannot overshoot $c_0(a)$ by more than half the distance between the histogram counts at a. `histeq` uses this transformation to map the gray levels in `X` (or the colormap) to their new values.

$$b = T(a)$$

If you do not specify `hgram`, `histeq` creates a flat `hgram`,

```
hgram = ones(1,n)*prod(size(A))/n;
```

and then applies the previous algorithm.

See Also

`brighten`, `imadjust`, `imhist`

Purpose

Convert hue-saturation-value (HSV) values to RGB color space

`hsv2rgb` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

idct2

Purpose Compute two-dimensional inverse discrete cosine transform

Syntax

```
B = idct2(A)
B = idct2(A,m,n)
B = idct2(A,[m n])
```

Description `B = idct2(A)` returns the two-dimensional inverse discrete cosine transform (DCT) of `A`.

`B = idct2(A,m,n)` or `B = idct2(A,[m n])` pads `A` with zeros to size `m`-by-`n` before transforming. If `[m n]` < `size(A)`, `idct2` crops `A` before transforming.

For any `A`, `idct2(dct2(A))` equals `A` to within roundoff error.

Class Support The input matrix `A` can be of class `double` or of any numeric class. The output matrix `B` is of class `double`.

Algorithm `idct2` computes the two-dimensional inverse DCT using

$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le m \le M-1 \\ 0 \le n \le N-1 \end{matrix}$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases} \qquad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

See Also `dct2`, `dctmtx`, `fft2`, `ifft2`

References

[1] Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. pp. 150-153.

[2] Pennebaker, William B., and Joan L. Mitchell. *JPEG: Still Image Data Compression Standard.* New York: Van Nostrand Reinhold, 1993.

Purpose

Compute two-dimensional inverse fast Fourier transform

`ifft2` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference pages.

Purpose Compute N-dimensional inverse fast Fourier transform

`ifftn` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference pages.

Purpose Convert an image to a binary image, based on threshold

Syntax

```
BW = im2bw(I,level)
BW = im2bw(X,map,level)
BW = im2bw(RGB,level)
```

Description `im2bw` produces binary images from indexed, intensity, or RGB images. To do this, it converts the input image to grayscale format (if it is not already an intensity image), and then converts this grayscale image to binary by thresholding. The output binary image `BW` has values of 0 (black) for all pixels in the input image with luminance less than `level` and 1 (white) for all other pixels. (Note that you specify `level` in the range [0,1], regardless of the class of the input image.)

`BW = im2bw(I,level)` converts the intensity image `I` to black and white.

`BW = im2bw(X,map,level)` converts the indexed image `X` with colormap `map` to black and white.

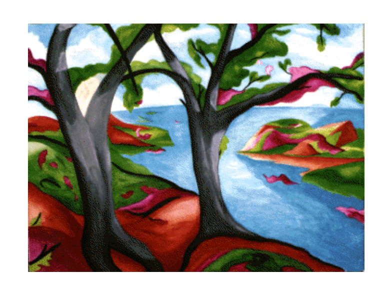
`BW = im2bw(RGB,level)` converts the RGB image `RGB` to black and white.

Note The function `graythresh` can be used to compute the level argument automatically.

Class Support The input image can be of class `uint8`, `uint16`, or `double` and it must be nonsparse. The output image, `BW`, is of class `logical`.

Example

```
load trees
BW = im2bw(X,map,0.4);
imshow(X,map)
figure, imshow(BW)
```





See Also graythresh, ind2gray, rgb2gray

Purpose Rearrange image blocks into columns

Syntax

```
B = im2col(A,[m n],block_type)
B = im2col(A,[m n])
B = im2col(A,'indexed',...)
```

Description `B = im2col(A,[m n],`*`block_type`*`)` rearranges image blocks into columns. *`block_type`* is a string that can have one of these values:

- `'distinct'` for `m`-by-`n` distinct blocks
- `'sliding'` for `m`-by-`n` sliding blocks (default)

`B = im2col(A,[m n],'distinct')` rearranges each distinct `m`-by-`n` block in the image `A` into a column of `B`. `im2col` pads `A` with zeros, if necessary, so its size is an integer multiple of `m`-by-`n`. If `A = [A11 A12;A21 A22]`, where each A_{ij} is `m`-by-`n`, then `B = [A11(:) A12(:) A21(:) A22(:)]`.

`B = im2col(A,[m n],'sliding')` converts each sliding `m`-by-`n` block of `A` into a column of `B`, with no zero padding. `B` has `m*n` rows and will contain as many columns as there are `m`-by-`n` neighborhoods of `A`. If the size of `A` is `[mm nn]`, then the size of `B` is (`m*n`)-by-`((mm-m+1)*(nn-n+1))`.

`B = im2col(A,[m n])` uses the default `block_type` of `'sliding'`.

For the sliding block case, each column of `B` contains the neighborhoods of `A` reshaped as `nhood(:)` where `nhood` is a matrix containing an `m`-by-`n` neighborhood of `A`. `im2col` orders the columns of `B` so that they can be reshaped to form a matrix in the normal way. For example, suppose you use a function, such as `sum(B)`, that returns a scalar for each column of `B`. You can directly store the result in a matrix of size (`mm-m+1`)-by-(`nn-n+1`), using these calls.

```
B = im2col(A,[m n],'sliding');
C = reshape(sum(B),mm-m+1,nn-n+1);
```

`B = im2col(A,'indexed',...)` processes `A` as an indexed image, padding with zeros if the class of `A` is `uint8`, or ones if the class of `A` is `double`.

Class Support The input image, `A`, can be numeric or logical. The output matrix, `B`, is of the same class as the input image.

See Also `blkproc`, `col2im`, `colfilt`, `nlfilter`

im2double

Purpose Convert image array to double precision

Syntax

```
I2 = im2double(I1)
RGB2 = im2double(RGB1)
I = im2double(BW)
X2 = im2double(X1,'indexed')
```

Description

`im2double` takes an image as input, and returns an image of class `double`. If the input image is of class `double`, the output image is identical to it. If the input image is of class `logical`, `uint8`, or `uint16`, `im2double` returns the equivalent image of class `double`, rescaling or offsetting the data as necessary.

`I2 = im2double(I1)` converts the intensity image `I1` to double precision, rescaling the data if necessary.

`RGB2 = im2double(RGB1)` converts the truecolor image `RGB1` to double precision, rescaling the data if necessary.

`I = im2double(BW)` converts the binary image `BW` to an intensity image of class `double`.

`X2 = im2double(X1,'indexed')` converts the indexed image `X1` to double precision, offsetting the data if necessary.

See Also `double`, `im2uint8`, `uint8`

Purpose Convert image to Java image

`im2java` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference pages.

im2uint8

Purpose Convert image array to eight-bit unsigned integers

Syntax

```
I2 = im2uint8(I1)
RGB2 = im2uint8(RGB1)
I = im2uint8(BW)
X2 = im2uint8(X1,'indexed')
```

Description

`im2uint8` takes an image as input, and returns an image of class `uint8`. If the input image is of class `uint8`, the output image is identical to it. If the input image is of class `logical`, `uint16`, or `double`, `im2uint8` returns the equivalent image of class `uint8`, rescaling or offsetting the data as necessary.

`I2 = im2uint8(I1)` converts the intensity image `I1` to `uint8`, rescaling the data if necessary.

`RGB2 = im2uint8(RGB1)` converts the truecolor image `RGB1` to `uint8`, rescaling the data if necessary.

`I = im2uint8(BW)` converts the binary image `BW` to an intensity image of class `uint8`.

`X2 = im2uint8(X1,'indexed')` converts the indexed image `X1` to `uint8`, offsetting the data if necessary. Note that it is not always possible to convert an indexed image to `uint8`. If `X1` is of class `double`, `max(X1(:))` must be 256 or less; if `X1` is of class `uint16`, `max(X1(:))` must be 255 or less. To convert a `uint16` indexed image to `uint8` by reducing the number of colors, use `imapprox`.

See Also `im2uint16`, `double`, `im2double`, `uint8`, `imapprox`, `uint16`

Purpose

Convert image array to 16-bit unsigned integers

Syntax

```
I2 = im2uint16(I1)
RGB2 = im2uint16(RGB1)
I = im2uint16(BW)
X2 = im2uint16(X1,'indexed')
```

Description

`im2uint16` takes an image as input, and returns an image of class `uint16`. If the input image is of class `uint16`, the output image is identical to it. If the input image is of class `double` or `uint8`, `im2uint16` returns the equivalent image of class `uint16`, rescaling or offsetting the data as necessary.

`I2 = im2uint16(I1)` converts the intensity image `I1` to `uint16`, rescaling the data if necessary.

`RGB2 = im2uint16(RGB1)` converts the truecolor image `RGB1` to `uint16`, rescaling the data if necessary.

`I = im2uint16(BW)` converts the binary image `BW` to an intensity image of class `uint16`.

`X2 = im2uint16(X1,'indexed')` converts the indexed image `X1` to `uint16`, offsetting the data if necessary. Note that it is not always possible to convert an indexed image to `uint16`. If `X1` is of class `double`, `max(X1(:))` must be 65536 or less.

Note `im2uint16` does not support binary images.

See Also

`im2uint8`, `double`, `im2double`, `uint8`, `uint16`, `imapprox`

imabsdiff

Purpose Compute absolute difference of two images

Syntax

```
Z = imabsdiff(X,Y)
```

Description

`Z = imabsdiff(X,Y)` subtracts each element in array `Y` from the corresponding element in array `X` and returns the absolute difference in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same class and size. `Z` has the same class and size as `X` and `Y`. If `X` and `Y` are integer arrays, elements in the output that exceed the range of the integer type are truncated.

If `X` and `Y` are double arrays, you can use the expression `abs(X-Y)` instead of this function.

Examples

This example calculates the absolute difference between two `uint8` arrays. Note that the absolute value prevents negative values from being rounded to zero in the result, as they are with `imsubtract`.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imabsdiff(X,Y)

Z =
   205    40    25
     6   175    50
```

Display the absolute difference between a filtered image and the original.

```
I = imread('cameraman.tif');
J = uint8(filter2(fspecial('gaussian'), I));
K = imabsdiff(I,J);
imshow(K,[]) % [] = scale data automatically
```

See also

`imadd`, `imcomplement`, `imdivide`, `imlincomb`, `immultiply`, `imsubtract`

Purpose

Add two images, or add a constant to an image

Syntax

```
Z = imadd(X,Y)
```

Description

`Z = imadd(X,Y)` adds each element in array `X` with the corresponding element in array `Y` and returns the sum in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same size and class, or `Y` is a scalar double. The array returned, `Z`, has the same size and class, or `Y` is a scalar double. `Z` has the same size and class as `X`.

If `X` and `Y` are integer arrays, elements in the output that exceed the range of the integer type are truncated, and fractional values are rounded.

If `X` and `Y` are double arrays, you can use the expression `X+Y` instead of this function.

Examples

Add two `uint8` arrays. Note the truncation that occurs when the values exceed 255.

```
X = uint8([ 255 0 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imadd(X,Y)
Z =

   255    50   125
    94   255   150
```

Add two images together and specify an output class.

```
I = imread('rice.tif');
J = imread('cameraman.tif');
K = imadd(I,J,'uint16');
imshow(K,[])
```

Add a constant to an image.

```
I = imread('rice.tif');
J = imadd(I,50);
subplot(1,2,1), imshow(I)
subplot(1,2,2), imshow(J)
```

See also

`imabsdiff`, `imcomplement`, `imdivide`, `imlincomb`, `immultiply`, `imsubtract`

Purpose Adjust image intensity values or colormap

Syntax

```
J = imadjust(I,[low_in high_in],[low_out high_out],gamma)
newmap = imadjust(map,[low_in high_in],[low_out high_out],gamma)
RGB2 = imadjust(RGB1,...)
```

Description

`J = imadjust(I,[low_in high_in],[low_out high_out],gamma)` maps the values in intensity image `I` to new values in `J` such that values between `low_in` and `high_in` map to values between `low_out` and `high_out`. Values below `low_in` and above `high_in` are clipped; that is, values below `low_in` map to `low_out`, and those above `high_in` map to `high_out`. You can use an empty matrix (`[]`) for [`low_in high_in`] or for [`low_out high_out`] to specify the default of `[0 1]`.

`gamma` specifies the shape of the curve describing the relationship between the values in `I` and `J`. If `gamma` is less than 1, the mapping is weighted toward higher (brighter) output values. If `gamma` is greater than 1, the mapping is weighted toward lower (darker) output values. If you omit the argument, `gamma` defaults to 1 (linear mapping).

`newmap = imadjust(map,[low_in; high_in],[low_out;high_out],gamma)` transforms the colormap associated with an indexed image. If `low_in`, `high_in`, `low_out`, `high_out`, and `gamma` are scalars, then the same mapping applies to red, green and blue components. Unique mappings for each color component are possible when

- `low_in` and `high_in` are both 1-by-3 vectors
- `low_out` and `high_out` are both 1-by-3 vectors, or `gamma` is a 1-by-3 vector.

The rescaled colormap, `newmap`, is the same size as `map`.

`RGB2 = imadjust(RGB1,...)` performs the adjustment on each image plane (red, green, and blue) of the RGB image `RGB1`. As with the colormap adjustment, you can apply unique mappings to each plane.

Note If `high_out < low_out`, the output image is reversed, as in a photographic negative.

The function `stretchlim` can be used with `imadjust` to apply an automatically computed contrast stretch.

Class Support

For syntax variations that include an input image (rather than a colormap), the input image can be of class `uint8`, `uint16`, or `double`. The output image has the same class as the input image. For syntax variations that include a colormap, the input and output colormaps are of class `double`.

Example

```
I = imread('pout.tif');
J = imadjust(I,[0.3 0.7],[]);
imshow(I), figure, imshow(J)
```





```
RGB1 = imread('flowers.tif');
RGB2 = imadjust(RGB1,[.2 .3 0; .6 .7 1],[]);
imshow(RGB1), figure, imshow(RGB2)
```





See Also `brighten`, `histeq`, `stretchlim`

imapprox

Purpose Approximate indexed image by one with fewer colors

Syntax

```
[Y,newmap] = imapprox(X,map,n)
[Y,newmap] = imapprox(X,map,tol)
Y = imapprox(X,map,newmap)
[...] = imapprox(...,dither_option)
```

Description

`[Y,newmap] = imapprox(X,map,n)` approximates the colors in the indexed image `X` and associated colormap `map` by using minimum variance quantization. `imapprox` returns indexed image `Y` with colormap `newmap`, which has at most `n` colors.

`[Y,newmap] = imapprox(X,map,tol)` approximates the colors in `X` and `map` through uniform quantization. `newmap` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0 and 1.0.

`Y = imapprox(X,map,newmap)` approximates the colors in `map` by using colormap mapping to find the colors in `newmap` that best match the colors in `map`.

`Y = imapprox(...,`*`dither_option`*`)` enables or disables dithering. *`dither_option`* is a string that can have one of these values:

- `'dither'` dithers, if necessary, to achieve better color resolution at the expense of spatial resolution (default).
- `'nodither'` maps each color in the original image to the closest color in the new map. No dithering is performed.

Class Support

The input image, `X`, can be of class `uint8`, `uint16`, or `double`. The output image `Y` is of class `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `X` is of class `double`.

Algorithm

`imapprox` uses `rgb2ind` to create a new colormap that uses fewer colors.

See Also

`cmunique`, `dither`, `rgb2ind`

imbothat

Purpose

Perform bottom-hat filtering

Syntax

```
IM2 = imbothat(IM,SE)
IM2 = imbothat(IM,NHOOD)
```

Description

`IM2 = imbothat(IM,SE)` performs morphological bottom-hat filtering on the grayscale or binary input image, `IM`, returning the filtered image, `IM2`. The argument `SE` is a structuring element returned by the `strel` function. `SE` must be a single structuring element object, not an array containing multiple structuring element objects.

`IM2 = imbothat(IM,NHOOD)` performs morphological bottom hat filtering where `NHOOD` is an array of 0's and 1's that specifies the size and shape of the structuring element. This is equivalent to `imbothat(IM,strel(NHOOD))`.

Class Support

`IM` can be numeric or logical and must be nonsparse. The output image has the same class as the input image. If the input is binary (logical), then the structuring element must be flat.

Example

Top-hat and bottom-hat filtering can be used together to enhance contrast in an image.

1 Read the image into the MATLAB workspace.

```
I = imread('pout.tif');
imshow(I), title('Original')
```



2 Add the original image to the top-hat filtered image, and then subtract the bottom-hat filtered image.

```
se = strel('disk',3);
J = imsubtract(imadd(I,imtophat(I,se)), imbothat(I,se));
figure, imshow(J), title('Contrast filtered')
```



See Also

imtophat, strel

imclearborder

Purpose Suppress light structures connected to image border

Syntax

```
IM2 = imclearborder(IM)
IM2 = imclearborder(IM,CONN)
```

Description `IM2 = imclearborder(IM)` suppresses structures that are lighter than their surroundings and that are connected to the image border. `IM` can be an intensity or binary image. The output image, `IM2`, is intensity or binary, respectively. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW),'maximal')` for higher dimensions.

Note For intensity images, `imclearborder` tends to reduce the overall intensity level in addition to suppressing border structures.

`IM2 = imclearborder(IM,CONN)` specifies the desired connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may also be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ... -by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Note A pixel on the edge of the input image might not be considered to be a "border" pixel if a nondefault connectivity is specified. For example, if `conn = [0 0 0; 1 1 1; 0 0 0]`, elements on the first and last row are not considered to be border pixels because, according to that connectivity definition, they are not connected to the region outside of the image.

Class Support

`IM` can be a numeric or logical array of any dimension, and it must be nonsparse and real. `IM2` has the same class as `IM`.

Example

The following examples use this simple binary image to illustrate the effect of of `imclearborder` when you specify different connectivities.

```
BW =
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     1     0     0     1     1     1     0     0     0
     0     1     0     1     1     1     0     0     0
     0     0     0     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
```

Using a 4-connected neighborhood, the pixel at `(5,2)` is not considered connected to the border pixel `(4,1)`, so it is not cleared.

```
BWc1 = imclearborder(BW,4)
BWc1 =
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     1     1     1     0     0     0
     0     1     0     1     1     1     0     0     0
     0     0     0     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
```

Using an 8-connected neighborhood, pixel (5,2) is considered connected to pixel (4,1) so both are cleared.

```
BWc2 = imclearborder(BW,8)

BWc2 =

     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     1     1     1     0     0     0
     0     0     0     1     1     1     0     0     0
     0     0     0     1     1     1     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
```

Algorithm

`imclearborder` uses morphological reconstruction where:

- the mask image is the input image
- the marker image is zero everywhere except along the border, where it equals the mask image

See Also

`conndef`

Reference

[1] P. Soille, *Morphological Image Analysis: Principles and Applications*, Springer, 1999, pp. 164-165.

Purpose Close an image

Syntax

```
IM2 = imclose(IM,SE)
IM2 = imclose(IM,NHOOD)
```

Description `IM2 = imclose(IM,SE)` performs morphological closing on the grayscale or binary image `IM`, returning the closed image, `IM2`. The structuring element, `SE`, must be a single structuring element object, as opposed to an array of objects.

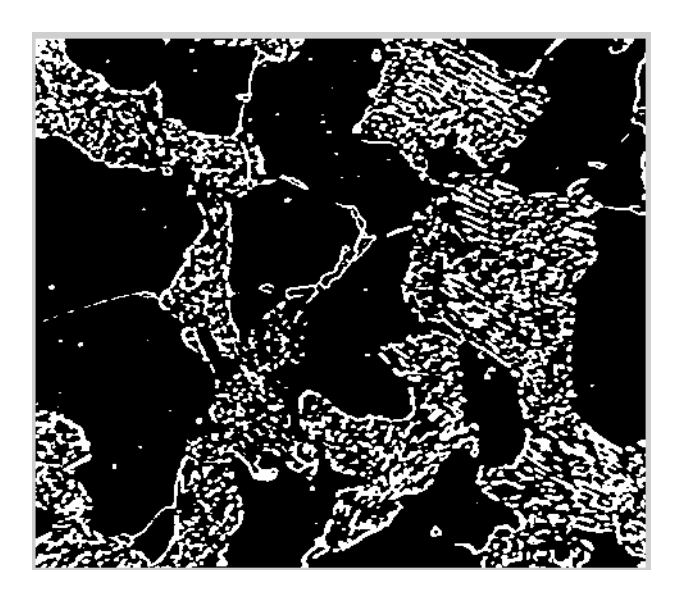
`IM2 = imclose(IM,NHOOD)` performs closing with the structuring element `strel(NHOOD)`, where `NHOOD` is an array of 0's and 1's that specifies the structuring element neighborhood.

Class Support `IM` can be any numeric or logical class and any dimension, and must be nonsparse. If `IM` is logical, then `SE` must be flat. `IM2` has the same class as `IM`.

Example This examples closes an image to merge together the small features in the image that are close together. The example then opens the image to remove the isolated white pixels.

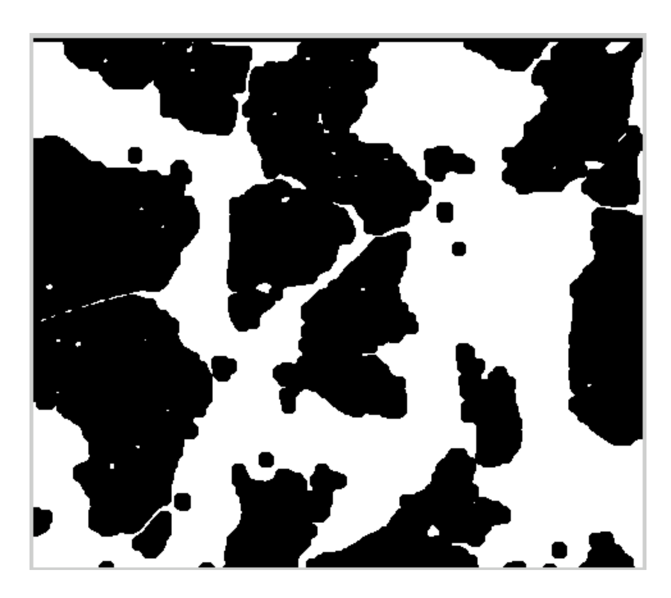
1 Read the image into the MATLAB workspace and threshold it.

```
I = imread('pearlite.tif');
bw = ~im2bw(I,graythresh(I));
figure, imshow(bw), title('Step 1: threshold')
```



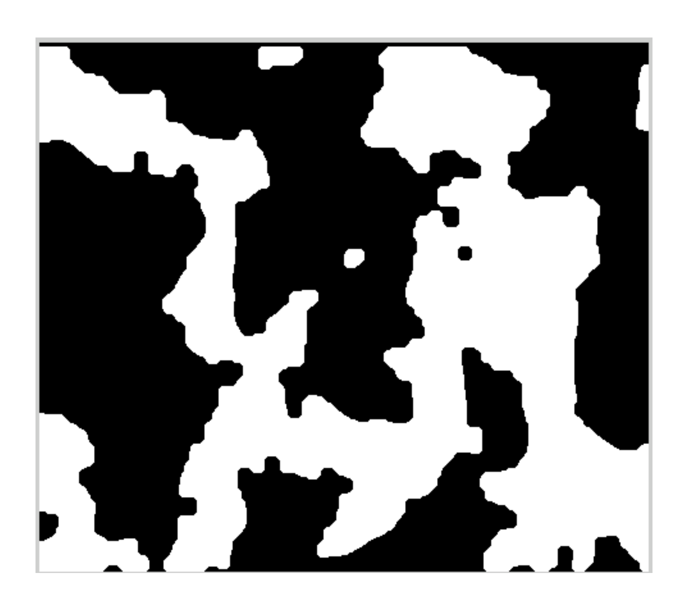
2 Close the image with a a disk-shaped structuring element.

```
se = strel('disk',6);
bw2 = imclose(bw,se);
figure, imshow(bw2), title('Step 2: closing')
```



3 Open the image with the same structuring element.

```
bw3 = imopen(bw2,se);
figure, imshow(bw3), title('Step 3: opening')
```



See Also

`imopen`, `imdilate`, `imerode`, `strel`

Purpose Complement image

Syntax

```
IM2 = imcomplement(IM)
```

Description

`IM2 = imcomplement(IM)` computes the complement of the image `IM`. `IM` can be a binary, intensity, or RGB image. `IM2` has the same class and size as `IM`.

In the complement of a binary image, zeros becomes ones and ones become zeros; black and white are reversed. In the complement of an intensity or RGB image, each pixel value is subtracted from the maximum pixel value supported by the class (or 1.0 for double-precision images) and the difference is used as the pixel value in the output image. In the output image, dark areas become lighter and light areas become darker.

Note If `IM` is an intensity or RGB image of class `double`, you can use the expression `1-IM` instead of this function. If `IM` is a binary image, you can use the expression `~IM` instead of this function.

Examples

Create the complement of a `uint8` array.

```
X = uint8([ 255 10 75; 44 225 100]);
X2 = imcomplement(X)
X2 =
     0   245   180
   211    30   155
```

Reverse black and white in a binary image.

```
bw = imread('text.tif');
bw2 = imcomplement(bw);
subplot(1,2,1),imshow(bw)
subplot(1,2,2),imshow(bw2)
```

Create the complement of an intensity image.

```
I = imread('bonemarr.tif');
J = imcomp(I);
imshow(I), figure, imshow(J)
```

See Also `imabsdiff`, `imadd`, `imdivide`, `imlincomb`, `immultiply`, `imsubtract`

Purpose Create a contour plot of image data

Syntax

```
imcontour(I,n)
imcontour(I,v)
imcontour(x,y,...)
imcontour(...,LineSpec)
[C,h] = imcontour(...)
```

Description

`imcontour(I,n)` draws a contour plot of the intensity image `I`, automatically setting up the axes so their orientation and aspect ratio match the image. `n` is the number of equally spaced contour levels in the plot; if you omit the argument, the number of levels and the values of the levels are chosen automatically.

`imcontour(I,v)` draws a contour plot of `I` with contour lines at the data values specified in vector `v`. The number of contour levels is equal to `length(v)`.

`imcontour(x,y,...)` uses the vectors `x` and `y` to specify the x- and y-axis limits.

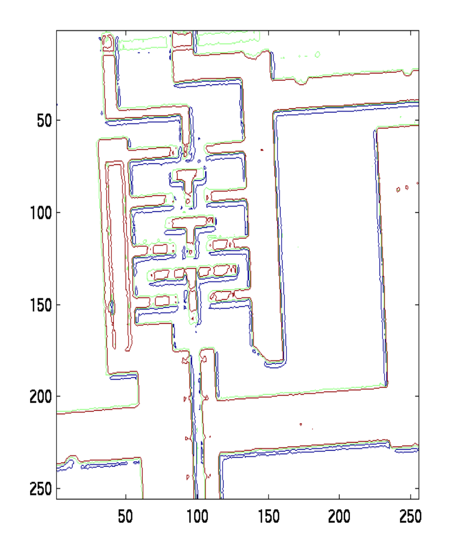
`imcontour(...,LineSpec)` draws the contours using the line type and color specified by `LineSpec`. Marker symbols are ignored.

`[C,h] = imcontour(...)` returns the contour matrix `C` and a vector of handles to the objects in the plot. (The objects are actually patches, and the lines are the edges of the patches.) You can use the `clabel` function with the contour matrix `C` to add contour labels to the plot.

Class Support The input image can be of class `uint8`, `uint16`, `double`, or `logical`.

Example

```
I = imread('ic.tif');
imcontour(I,3)
```



See Also

`clabel`, `contour`, `LineSpec` in the MATLAB Function Reference

Purpose

Crop an image

Syntax

```
I2 = imcrop(I)
X2 = imcrop(X,map)
RGB2 = imcrop(RGB)

I2 = imcrop(I,rect)
X2 = imcrop(X,map,rect)
RGB2 = imcrop(RGB,rect)

[...] = imcrop(x,y,...)
[A,rect] = imcrop(...)
[x,y,A,rect] = imcrop(...)
```

Description

`imcrop` crops an image to a specified rectangle. In the syntaxes below, `imcrop` displays the input image and waits for you to specify the crop rectangle with the mouse.

```
I2 = imcrop(I)
X2 = imcrop(X,map)
RGB2 = imcrop(RGB)
```

If you omit the input arguments, `imcrop` operates on the image in the current axes.

To specify the rectangle:

- For a single-button mouse, press the mouse button and drag to define the crop rectangle. Finish by releasing the mouse button.
- For a 2- or 3-button mouse, press the left mouse button and drag to define the crop rectangle. Finish by releasing the mouse button.

If you hold down the **Shift** key while dragging, or if you press the right mouse button on a 2- or 3-button mouse, `imcrop` constrains the bounding rectangle to be a square.

When you release the mouse button, `imcrop` returns the cropped image in the supplied output argument. If you do not supply an output argument, `imcrop` displays the output image in a new figure.

You can also specify the cropping rectangle noninteractively, using these syntaxes

```
I2 = imcrop(I,rect)
X2 = imcrop(X,map,rect)
RGB2 = imcrop(RGB,rect)
```

`rect` is a four-element vector with the form `[xmin ymin width height]`; these values are specified in spatial coordinates.

To specify a nondefault spatial coordinate system for the input image, precede the other input arguments with two, two-element vectors specifying the `XData` and `YData`. For example,

```
[...] = imcrop(x,y,...)
```

If you supply additional output arguments, `imcrop` returns information about the selected rectangle and the coordinate system of the input image. For example,

```
[A,rect] = imcrop(...)
[x,y,A,rect] = imcrop(...)
```

`A` is the output image. `x` and `y` are the `XData` and `YData` of the input image.

Class Support

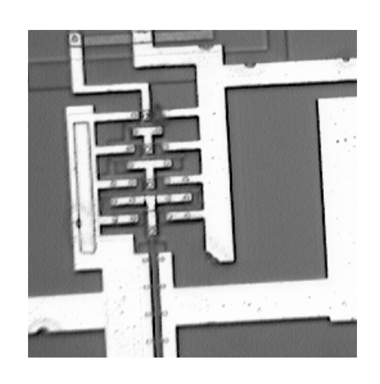
The input image `A` can be of class `logical`, `uint8`, `uint16`, or `double`. The output image `B` is of the same class as `A`. `rect` is always of class `double`.

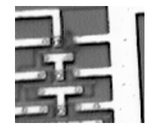
Remarks

Because `rect` is specified in terms of spatial coordinates, the `width` and `height` elements of `rect` do not always correspond exactly with the size of the output image. For example, suppose `rect` is [20 20 40 30], using the default spatial coordinate system. The upper-left corner of the specified rectangle is the center of the pixel (20,20) and the lower-right corner is the center of the pixel (50,60). The resulting output image is 31-by-41, not 30-by-40, because the output image includes all pixels in the input image that are completely *or partially* enclosed by the rectangle.

Example

```
I = imread('ic.tif');
I2 = imcrop(I,[60 40 100 90]);
imshow(I)
figure, imshow(I2)
```





See Also

`zoom`

Purpose Dilate image

Syntax

```
IM2 = imdilate(IM,SE)
IM2 = imdilate(IM,NHOOD)
IM2 = imdilate(IM,SE,PACKOPT)
IM2 = imdilate(...,PADOPT)
```

Description

`IM2 = imdilate(IM,SE)` dilates the grayscale, binary, or packed binary image `IM`, returning the dilated image, `IM2`. The argument `SE` is a structuring element object, or array of structuring element objects, returned by the `strel` function.

If `IM` is logical and the structuring element is flat, `imdilate` performs binary dilation; otherwise, it performs grayscale dilation. If `SE` is an array of structuring element objects, `imdilate` performs multiple dilations of the input image, using each structuring element in `SE` in succession.

`IM2 = imdilate(IM,NHOOD)` dilates the image `IM`, where `NHOOD` is a matrix of 0's and 1's that specifies the structuring element neighborhood. This is equivalent to the syntax `imdilate(IM,strel(NHOOD))`. The `imdilate` function determines the center element of the neighborhood by `floor((size(NHOOD)+1)/2)`.

`IM2 = imdilate(IM,SE,PACKOPT)` or `imdilate(IM,NHOOD,PACKOPT)` specifies whether `IM` is a packed binary image. `PACKOPT` can have either of the following values.

`'ispacked'`	`IM` is treated as a packed binary image as produced by `bwpack`. `IM` must be a 2-D `uint32` array and `SE` must be a flat 2-D structuring element. If the value of `PACKOPT` is `'ispacked'`, `PADOPT` must be `'same'`.
`'notpacked'`	`IM` is treated as a normal array. This is the default value.

`IM2 = imdilate(...,PADOPT)` specifies the size of the output image. `PADOPT` can have either of the following values.

`'same'`	Make the output image the same size as the input image. This is the default value. If the value of `PACKOPT` is `'ispacked'`, `PADOPT` must be `'same'`.
`'full'`	Compute the full dilation.

`PADOPT` is analogous to the optional `SHAPE` argument to the `conv2` and `filter2` functions.

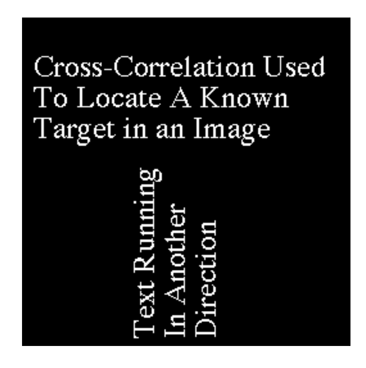
Class Support

`IM` can be logical or numeric and must be real and nonsparse. It can have any dimension. If `IM` is logical, `SE` must be flat. The output has the same class as the input. If the input is packed binary, then the output is also packed binary.

Examples

This example dilates a binary image with a vertical line structuring element.

```
bw = imread('text.tif');
se = strel('line',11,90);
bw2 = imdilate(bw,se);
imshow(bw), title('Original')
figure, imshow(bw2), title('Dilated')
```





This example dilates a grayscale image with a rolling ball structuring element.

```
I = imread('cameraman.tif');
se = strel('ball',5,5);
```

```
I2 = imdilate(I,se);
imshow(I), title('Original')
figure, imshow(I2), title('Dilated')
```

To determine the domain of the composition of two flat structuring elements, dilate the scalar value 1 with both structuring elements in sequence, using the 'full' option.

```
se1 = strel('line',3,0)
se1 =

Flat STREL object containing 3 neighbors.
Neighborhood:
     1     1     1

se2 = strel('line',3,90)
se2 =

Flat STREL object containing 3 neighbors.
Neighborhood:
     1
     1
     1

composition = imdilate(1,[se1 se2],'full')
composition =
     1     1     1
     1     1     1
```

```
        1         1         1
```

Algorithm

`imdilate` automatically takes advantage of the decomposition of a structuring element object (if it exists). Also, when performing binary dilation with a structuring element object that has a decomposition, `imdilate` automatically uses binary image packing to speed up the dilation.

Dilation using bit packing is described in [2].

See Also

`bwpack`, `bwunpack`, `conv2`, `filter2`, `imclose`, `imerode`, `imopen`, `strel`

References

[1] Robert M. Haralick and Linda G. Shapiro, *Computer and Robot Vision*, vol. I, Addison-Wesley, 1992, pp. 158-205.

[2] van den Boomgaard and van Balen, “Image Transforms Using Bitmapped Binary Images,” *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, vol. 54, no. 3, May, 1992, pp. 254-258.

imdivide

Purpose Divide one image into another, or divide an image by a constant

Syntax

```
Z = imdivide(X,Y)
```

Description `Z = imdivide(X,Y)` divides each element in the array `X` by the corresponding element in array `Y` and returns the result in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same size and class, or `Y` can be a scalar double. `Z` has the same size and class as `X` and `Y`.

If `X` is an integer array, elements in the output that exceed the range of integer type are truncated, and fractional values are rounded.

If `X` and `Y` are double arrays, you can use the expression `X./Y` instead of this function.

Example Divide two `uint8` arrays. Note that fractional values greater than or equal to 0.5 are rounded up to the nearest integer.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 20 50; 50 50 50 ]);
Z = imdivide(X,Y)
Z =
     5     1     2
     1     5     2
```

Estimate and divide out the background of the rice image.

```
I = imread('rice.tif');
blocks = blkproc(I,[32 32],'min(x(:))');
background = imresize(blocks,[256 256],'bilinear');
Ip = imdivide(I,background);
imshow(Ip,[]) % [] = let imshow scale data automatically
```

Divide an image by a constant factor.

```
I = imread('rice.tif');
J = imdivide(I,2);
subplot(1,2,1), imshow(I)
subplot(1,2,2), imshow(J)
```

See Also `imabsdiff`, `imadd`, `imcomplement`, `imlincomb`, `immultiply`, `imsubtract`

Purpose

Erode image

Syntax

```
IM2 = imerode(IM,SE)
IM2 = imerode(IM,NHOOD)
IM2 = imerode(IM,SE,PACKOPT,M)
IM2 = imerode(...,PADOPT)
```

Description

`IM2 = imerode(IM,SE)` erodes the grayscale, binary, or packed binary image `IM`, returning the eroded image, `IM2`. The argument `SE` is a structuring element object, or array of structuring element objects, returned by the `strel` function.

If `IM` is logical and the structuring element is flat, `imerode` performs binary dilation; otherwise it performs grayscale erosion. If `SE` is an array of structuring element objects, `imerode` performs multiple erosions of the input image, using each structuring element in `SE` in succession.

`IM2 = imerode(IM,NHOOD)` erodes the image `IM`, where `NHOOD` is an array of `0`'s and `1`'s that specifies the structuring element neighborhood. This is equivalent to the syntax `imerode(IM,strel(NHOOD))`. The `imerode` function determines the center element of the neighborhood by `floor((size(NHOOD)+1)/2)`

`IM2 = imerode(IM,SE,PACKOPT,M)` or `imerode(IM,NHOOD,PACKOPT,M)` specifies whether `IM` is a packed binary image and, if it is, provides the row dimension, `M`, of the original unpacked image. `PACKOPT` can have either of the following values.

`'ispacked'`	`IM` is treated as a packed binary image as produced by `bwpack`. `IM` must be a 2-D `uint32` array and `SE` must be a flat 2-D structuring element.
`'notpacked'`	`IM` is treated as a normal array. This is the default value.

If `PACKOPT` is `'ispacked'`, you must specify a value for `M`.

`IM2 = imerode(...,PADOPT)` specifies the size of the output image. `PADOPT` can have either of the following values.

`'same'`	Make the output image the same size as the input image. This is the default value. If the value of `PACKOPT` is `'ispacked'`, `PADOPT` must be `'same'`.
`'full'`	Compute the full erosion.

`PADOPT` is analogous to the `SHAPE` input to the `CONV2` and `FILTER2` functions.

Class Support

`IM` can be numeric or logical and it can be of any dimension. If `IM` is logical, `SE` must be flat. The output has the same class as the input. If the input is packed binary, then the output is also packed binary.

Examples

This example erodes a binary image with a vertical line.

```
bw = imread('text.tif');
se = strel('line',11,90);
bw2 = imerode(bw,se);
imshow(bw), title('Original')
figure, imshow(bw2), title('Eroded')
```



This example erodes a grayscale image with a rolling ball.

```
I = imread('cameraman.tif');
se = strel('ball',5,5);
I2 = imerode(I,se);
```

```
imshow(I), title('Original')
figure, imshow(I2), title('Eroded')
```




Algorithm Notes

`imerode` automatically takes advantage of the decomposition of a structuring element object (if a decomposition exists). Also, when performing binary dilation with a structuring element object that has a decomposition, `imerode` automatically uses binary image packing to speed up the dilation.

Erosion using bit packing is described in [2].

See Also

`bwpack`, `bwunpack`, `conv2`, `filter2`, `imclose`, `imdilate`, `imopen`, `strel`

References

[1] Robert M. Haralick and Linda G. Shapiro, *Computer and Robot Vision*, vol. I, Addison-Wesley, 1992, pp. 158-205.

[2] van den Boomgaard and van Balen, “Image Transforms Using Bitmapped Binary Images,” *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, vol. 54, no. 3, May, 1992, pp. 254-258.

imextendedmax

Purpose Extended-maxima transform

Syntax

```
BW = imextendedmax(I,H)
```

Description

`BW = imextendedmax(I,H)` computes the extended-maxima transform, which is the regional maxima of the H-maxima transform. `H` is a nonnegative scalar.

Regional maxima are connected components of pixels with the same intensity value, *t*, whose external boundary pixels all have a value less than *t*.

By default, `imextendedmax` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imextendedmax` uses `conndef(ndims(I),'maximal')`.

`BW = imextendedmax(I,H,CONN)` computes the extended-maxima transform, where `CONN` specifies the connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

`I` can be of any nonsparse numeric class and any dimension. `BW` has the same size as `I` and is always logical.

Example

```
I = imread('bonemarr.tif');
BW = imextendedmax(I,40);
imshow(I), figure, imshow(BW)
```

See Also

`conndef`, `imextendedmin`, `imreconstruct`

Reference

[1] Pierre Soille, *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

imextendedmin

Purpose

Extended-minima transform

Syntax

```
BW = imextendedmin(I,h)
```

Description

`BW = imextendedmin(I,h)` computes the extended-minima transform, which is the regional minima of the H-minima transform. `h` is a nonnegative scalar.

Regional minima are connected components of pixels with the same intensity value, *t*, whose external boundary pixels all have a value greater than *t*.

By default, `imextendedmin` uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, `imextendedmin` uses `conndef(ndims(I),'maximal')`.

`BW = imextendedmin(I,h,CONN)` computes the extended-minima transform, where `CONN` specifies the connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by-...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

`I` can be of any nonsparse numeric class and any dimension. `BW` has the same size as `I` and is always logical.

Example

```
I = imread('bonemarr.tif');
BW = imextendedmin(I,10);
imshow(I), figure, imshow(BW)
```

See Also

`conndef`, `imextendedmax`, `imreconstruct`

Reference

[1] Pierre Soille, *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

Purpose Compute feature measurements for image regions

Note This function is obsolete and may be removed in future versions. Use `regionprops` instead.

Syntax

```
stats = imfeature(L,measurements)
stats = imfeature(L,measurements,n)
```

Description

`stats = imfeature(L,`*`measurements`*`)` computes a set of measurements for each labeled region in the label matrix `L`. Positive integer elements of `L` correspond to different regions. For example, the set of elements of `L` equal to 1 corresponds to region 1; the set of elements of `L` equal to 2 corresponds to region 2; and so on. `stats` is a structure array of length `max(L(:))`. The fields of the structure array denote different measurements for each region, as specified by *`measurements`*.

`measurements` can be a comma-separated list of strings, a cell array containing strings, the single string `'all'`, or the single string `'basic'`. The set of valid measurement strings includes the following.

`'Area'`	`'Image'`	`'EulerNumber'`
`'Centroid'`	`'FilledImage'`	`'Extrema'`
`'BoundingBox'`	`'FilledArea'`	`'EquivDiameter'`
`'MajorAxisLength'`	`'ConvexHull'`	`'Solidity'`
`'MinorAxisLength'`	`'ConvexImage'`	`'Extent'`
`'Eccentricity'`	`'ConvexArea'`	`'PixelList'`
`'Orientation'`		

Measurement strings are case insensitive and can be abbreviated.

If *`measurements`* is the string `'all'`, then all of the above measurements are computed. If *`measurements`* is not specified or if it is the string `'basic'`, then these measurements are computed: `'Area'`, `'Centroid'`, and `'BoundingBox'`.

`stats = imfeature(L,measurements,n)` specifies the type of connectivity used in computing the `'FilledImage'`, `'FilledArea'`, and `'EulerNumber'` measurements. `n` can have a value of either 4 or 8, where 4 specifies 4-connected objects and 8 specifies 8-connected objects; if the argument is omitted, it defaults to 8.

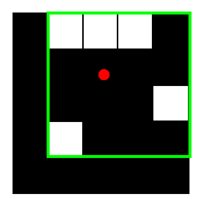
Definitions

`'Area'` – Scalar; the actual number of pixels in the region. (This value may differ slightly from the value returned by `bwarea`, which weights different patterns of pixels differently.)

`'Centroid'` – 1-by-2 vector; the *x*- and *y*-coordinates of the center of mass of the region.

`'BoundingBox'` – 1-by-4 vector; the smallest rectangle that can contain the region. The format of the vector is `[x y width height]`, where `x` and `y` are the *x*- and *y*-coordinates of the upper-left corner of the rectangle, and `width` and `height` are the width and height of the rectangle. Note that `x` and `y` are always noninteger values, because they are the spatial coordinates for the upper-left corner of a pixel in the image; for example, if this pixel is the third pixel in the fifth row of the image, then `x` = 2.5 and `y` = 4.5.

This figure illustrates the centroid and bounding box. The region consists of the white pixels; the green box is the bounding box, and the red dot is the centroid.



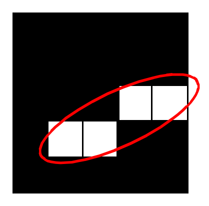
`'MajorAxisLength'` – Scalar; the length (in pixels) of the major axis of the ellipse that has the same second-moments as the region.

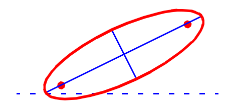
`'MinorAxisLength'` – Scalar; the length (in pixels) of the minor axis of the ellipse that has the same second-moments as the region.

`'Eccentricity'` – Scalar; the eccentricity of the ellipse that has the same second-moments as the region. The eccentricity is the ratio of the distance between the foci of the ellipse and its major axis length. The value is between 0 and 1. (0 and 1 are degenerate cases; an ellipse whose eccentricity is 0 is actually a circle, while an ellipse whose eccentricity is 1 is a line segment.)

`'Orientation'` – Scalar; the angle (in degrees) between the *x*-axis and the major axis of the ellipse that has the same second-moments as the region.

This figure illustrates the axes and orientation of the ellipse. The left side of the figure shows an image region and its corresponding ellipse. The right side shows the same ellipse, with features indicated graphically; the solid blue lines are the axes, the red dots are the foci, and the orientation is the angle between the horizontal dotted line and the major axis.



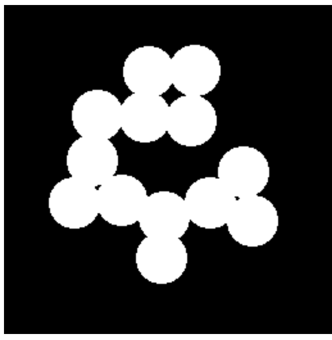


`'Image'` – Binary image of the same size as the bounding box of the region; the `on` pixels correspond to the region, and all other pixels are `off`.

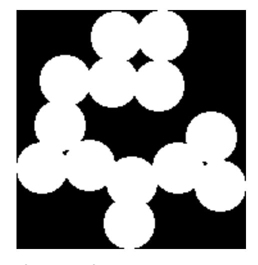
`'FilledImage'` – Binary image of the same size as the bounding box of the region; the `on` pixels correspond to the region, with all holes filled in.

`'FilledArea'` – Scalar; the number of `on` pixels in `FilledImage`.

This figure illustrates `'Image'` and `'FilledImage'`.



Original image, containing a single region



`'Image'`



`'FilledImage'`

'ConvexHull' – p-by-2 matrix; the smallest convex polygon that can contain the region. Each row of the matrix contains the *x*- and *y*-coordinates of one vertex of the polygon.

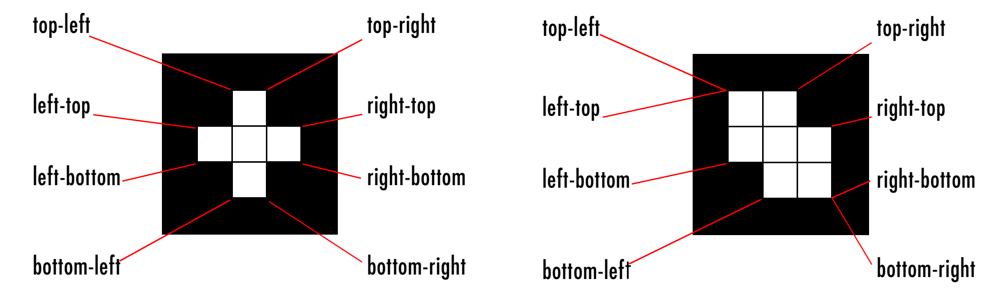
'ConvexImage' – Binary image (uint8); the convex hull, with all pixels within the hull filled in (i.e., set to on). (For pixels that the boundary of the hull passes through, imfeature uses the same logic as roipoly to determine whether the pixel is inside or outside the hull.) The image is the size of the bounding box of the region.

'ConvexArea' – Scalar; the number of pixels in 'ConvexImage'.

'EulerNumber' – Scalar; equal to the number of objects in the region minus the number of holes in those objects.

'Extrema' – 8-by-2 matrix; the extremal points in the region. Each row of the matrix contains the *x*- and *y*-coordinates of one of the points; the format of the vector is [top-left top-right right-top right-bottom bottom-right bottom-left left-bottom left-top].

This figure illustrates the extrema of two different regions. In the region on the left, each extremal point is distinct; in the region on the right, certain extremal points (e.g., top-left and left-top) are identical.



'EquivDiameter' – Scalar; the diameter of a circle with the same area as the region. Computed as sqrt(4*Area/pi).

'Solidity' – Scalar; the proportion of the pixels in the convex hull that are also in the region. Computed as Area/ConvexArea.

'Extent' – Scalar; the proportion of the pixels in the bounding box that are also in the region. Computed as the Area divided by area of the bounding box.

`'PixelList'` – p-by-2 matrix; the actual pixels in the region. Each row of the matrix contains the *x*- and *y*-coordinates of one pixel in the region.

Class Support

The input label matrix `L` can be of class `double` or of any integer class.

Remarks

The comma-separated list syntax for structure arrays is very useful when working with the output of `imfeature`. For example, for a field that contains a scalar, you can use a this syntax to create a vector containing the value of this field for each region in the image.

For instance, if `stats` is a structure array with field `Area`, then these two expressions are equivalent

```
stats(1).Area, stats(2).Area, ..., stats(end).Area
```

and

```
stats.Area
```

Therefore, you can use these calls to create a vector containing the area of each region in the image.

```
stats = imfeature(L,'Area');
allArea = [stats.Area];
```

`allArea` is a vector of the same length as the structure array `stats`.

The function `ismember` is useful in conjunction with `imfeature` for selecting regions based on certain criteria. For example, these commands create a binary image containing only the regions in `text.tif` whose area is greater than 80.

```
idx = find([stats.Area] > 80);
BW2 = ismember(L,idx);
```

Most of the measurements take very little time to compute. The exceptions are these, which may take significantly longer, depending on the number of regions in `L`:

- `'ConvexHull'`
- `'ConvexImage'`
- `'ConvexArea'`
- `'FilledImage'`

Note that computing certain groups of measurements takes about the same amount of time as computing just one of them, because `imfeature` takes advantage of intermediate computations used in both computations. Therefore, it is fastest to compute all of the desired measurements in a single call to `imfeature`.

Example

```
BW = imread('text.tif');
L = bwlabel(BW);
stats = imfeature(L,'all');
stats(23)

ans =

                Area: 89
            Centroid: [95.6742 192.9775]
         BoundingBox: [87.5000 184.5000 16 15]
     MajorAxisLength: 19.9127
     MinorAxisLength: 14.2953
        Eccentricity: 0.6961
         Orientation: 9.0845
          ConvexHull: [13x2  double]
         ConvexImage: [15x16 logical ]
          ConvexArea: 205
               Image: [15x16 logical ]
         FilledImage: [15x16 logical ]
          FilledArea: 122
         EulerNumber: 0
             Extrema: [ 8x2  double]
       EquivDiameter: 10.6451
            Solidity: 0.4341
              Extent: 0.3708
           PixelList: [89x2  double]
```

See Also

`bwlabel`

`ismember` in the MATLAB Function Reference

imfill

Purpose Fill image regions

Syntax

```
BW2 = imfill(BW)
[BW2,LOCATIONS] = imfill(BW)
BW2 = imfill(BW,LOCATIONS)
BW2 = imfill(BW,LOCATIONS,CONN)
BW2 = imfill(BW,'holes')
BW2 = imfill(BW,CONN,'holes')
I2 = imfill(I,'holes')
I2 = imfill(I,CONN,'holes')
```

Description

`BW2 = imfill(BW,LOCATIONS)` performs a flood-fill operation on background pixels of the input binary image `BW`, starting from the points specified in `LOCATIONS`. `LOCATIONS` can be a P-by-1 vector, in which case it contains the linear indices of the starting locations. `LOCATIONS` can also be a `P`-by-`ndims(BW)` matrix, in which case each row contains the array indices of one of the starting locations.

`BW2 = imfill(BW,'holes')` fills holes in the input binary image. A hole is a set of background pixels that cannot be reached by filling in the background from the edge of the image.

`I2 = imfill(I,'holes')` fills holes in an input intensity image, `I`. In this case, a hole is an area of dark pixels surrounded by lighter pixels.

Interactive Use

`BW2 = imfill(BW)` displays the binary image, `BW`, on the screen and lets you select the starting locations using the mouse. Click the mouse button to add points. Press `<BackSpace>` or `<Delete>` to remove the previously selected point. A shift-click, right-click, or double-click selects a final point and then starts the fill operation; pressing `<Return>` finishes the selection without adding a point. Interactive use is supported only for 2-D images.

The syntax `[BW2,LOCATIONS] = imfill(BW)` can be used to get the starting points selected using the mouse. The output `LOCATIONS` is a vector of linear indices into the input image.

Specifying Connectivity

By default, `imfill` uses 4-connected background neighbors for 2-D inputs and 6-connected background neighbors for 3-D inputs. For higher dimensions the

default background connectivity is determined by using `conndef(NUM_DIMS,'minimal')`.

You can override the default connectivity with these syntaxes:

```
BW2 = imfill(BW,LOCATIONS,CONN)
BW2 = imfill(BW,CONN,'holes')
I2  = imfill(I,CONN,'holes')
```

To override the default connectivity and interactively specify the starting locations, use this syntax:

```
BW2 = imfill(BW,0,CONN)
```

`CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
`4`	4-connected neighborhood
`8`	8-connected neighborhood
Three-dimensional connectivities	
`6`	6-connected neighborhood
`18`	18-connected neighborhood
`26`	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ... -by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

The input image can be numeric or logical, and it must be real and nonsparse. It can have any dimension. The output image has the same class as the input image.

Examples

Fill in the background of a binary image from a specified starting location:

```
BW1 = [1 0 0 0 0 0 0 0
       1 1 1 1 1 0 0 0
       1 0 0 0 1 0 1 0
       1 0 0 0 1 1 1 0
       1 1 1 1 0 1 1 1
       1 0 0 1 1 0 1 0
       1 0 0 0 1 0 1 0
       1 0 0 0 1 1 1 0]

BW2 = imfill(BW1,[3 3],8)
```

Fill in the holes of a binary image:

```
BW4 = ~im2bw(imread('blood1.tif'));
BW5 = imfill(BW4,'holes');
imshow(BW4), figure, imshow(BW5)
```

Fill in the holes of an intensity image:

```
I = imread('enamel.tif');
I2 = imcomplement(imfill(imcomplement(I),'holes'));
imshow(I), figure, imshow(I2)
```

Algorithm

`imfill` uses an algorithm based on morphological reconstruction [1].

See Also

`bwselect`, `imreconstruct`, `roifill`

Reference

[1] Pierre Soille, *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 173-174.

Purpose Multidimensional image filtering

Syntax

```
B = imfilter(A,H)
B = imfilter(A,H,option1,option2,...)
```

Description

`B = imfilter(A,H)` filters the multidimensional array `A` with the multidimensional filter `H`. The array, `A`, can be a nonsparse numeric array of any class and dimension. The result, `B`, has the same size and class as `A`.

Each element of the output, `B`, is computed using double-precision floating point. If `A` is an integer array, then output elements that exceed the range of the integer type are truncated, and fractional values are rounded.

`B = imfilter(A,H,option1,option2,...)` performs multidimensional filtering according to the specified options. Option arguments can have the following values.

Boundary Options

Option	Description
`X`	Input array values outside the bounds of the array are implicitly assumed to have the value `X`. When no boundary option is specified, `imfilter` uses `X = 0`.
`'symmetric'`	Input array values outside the bounds of the array are computed by mirror-reflecting the array across the array border.
`'replicate'`	Input array values outside the bounds of the array are assumed to equal the nearest array border value.
`'circular'`	Input array values outside the bounds of the array are computed by implicitly assuming the input array is periodic.

Output Size Options

Option	Description
`'same'`	The output array is the same size as the input array. This is the default behavior when no output size options are specified.
`'full'`	The output array is the full filtered result, and so is larger than the input array.

Correlation and Convolution Options

Option	Description
`'corr'`	`imfilter` performs multidimensional filtering using correlation, which is the same way that `filter2` performs filtering. When no correlation or convolution option is specified, `imfilter` uses correlation.
`'conv'`	`imfilter` performs multidimensional filtering using convolution.

N-D convolution is related to N-D correlation by a reflection of the filter matrix.

Examples

This example starts by reading an image of a bouquet of flowers into a three-dimensional array of `uint8` values, called `rgb`. The `imshow` function displays the original image, entitled `'Original'`.

```
rgb = imread('flowers.tif');
imshow(rgb), title('Original')
```

Create a filter, `h`, that can be used to approximate linear camera motion. Use `imfilter` on the three-dimensional RGB image, `rgb`, to create a new image, `rgb2`.

```
h = fspecial('motion', 50, 45);
rgb2 = imfilter(rgb, h);
figure, imshow(rgb2), title('Filtered')
```

Note that `imfilter` is more memory efficient than some other filtering operations in that it outputs an array of the same data type as the input image array. In this example, the output is an array of `uint8`.

```
whos rgb2
  Name       Size            Bytes  Class

  rgb2     362x500x3       543000  uint8 array
```

Filter the image once more, this time specifying the `replicate` boundary option and observe that the picture now extends to the border.

```
rgb3 = imfilter(rgb, h, 'replicate');
figure, imshow(rgb3), title('Filtered with boundary replication')
```

See Also

`conv2`, `convn`, `filter2`

Purpose

Information about graphics file

`imfinfo` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference pages.

Purpose Display a histogram of image data

Syntax

```
imhist(I,n)
imhist(X,map)
[counts,x] = imhist(...)
```

Description

`imhist(I,n)` displays a histogram with `n` bins for the intensity image `I` above a grayscale colorbar of length `n`. If you omit the argument, `imhist` uses a default value of `n` = 256 if `I` is a grayscale image, or `n` = 2 if `I` is a binary image.

`imhist(X,map)` displays a histogram for the indexed image `X`. This histogram shows the distribution of pixel values above a colorbar of the colormap `map`. The colormap must be at least as long as the largest index in `X`. The histogram has one bin for each entry in the colormap.

`[counts,x] = imhist(...)` returns the histogram counts in `counts` and the bin locations in `x` so that `stem(x,counts)` shows the histogram. For indexed images, it returns the histogram counts for each colormap entry; the length of `counts` is the same as the length of the colormap.

Note For intensity images, the `n` bins of the histogram are each half-open intervals of width $A/(n-1)$. In particular, the p th bin is the half-open interval

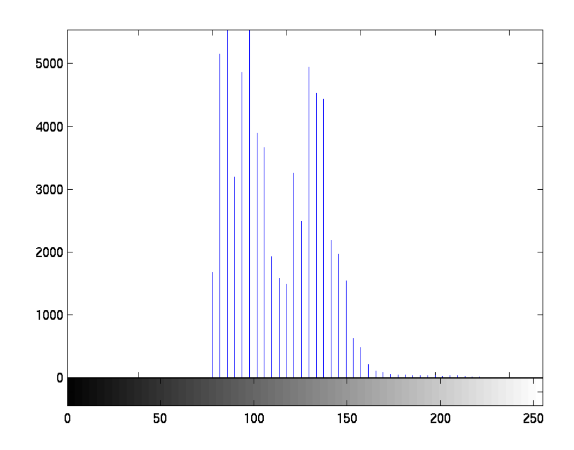
$$A(p-1.5)/(n-1) \leq x < A(p-0.5)/(n-1)$$

The scale factor A depends on the image class. A is 1 if the intensity image is `double`; A is 255 if the intensity image is `uint8`; and A is 65535 if the intensity image is `uint16`.

Class Support The input image can be of class `logical`, `uint8`, `uint16`, or `double`.

Example

```
I = imread('pout.tif');
imhist(I)
```



See Also

`histeq`

`hist` is in the MATLAB Function Reference

imhmax

Purpose H-maxima transform

Syntax

```
I2 = imhmax(I,h)
```

Description

`I2 = imhmax(I,h)` suppresses all maxima in the intensity image, `I`, whose height is less than `h`, where `h` is a scalar.

Regional maxima are connected components of pixels with the same intensity value, *t*, whose external boundary pixels all have a value less than *t*.

By default, `imhmax` uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmax` uses `conndef(ndims(I),'maximal')`.

`I2 = imhmax(I,h,CONN)` computes the H-maxima transform, where `CONN` specifies the connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

`I` can be of any nonsparse numeric class and any dimension. `I2` has the same size and class as `I`.

Example

```
a = zeros(10,10);
a(2:4,2:4) = 3;  % maxima 3 higher than surround
a(6:8,6:8) = 8;  % maxima 8 higher than surround
b = imhmax(a,4); % only the maxima higher than 4 survive.
```

See Also `conndef`, `imhmin`, `imreconstruct`

Reference

[1] Pierre Soille, *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

Purpose

H-minima transform

Syntax

```
I2 = imhmin(I,h)
```

Description

`I2 = imhmin(I,h)` suppresses all minima in the intensity image, `I`, whose depth is less than `h`, where `h` is a scalar.

Regional minima are connected components of pixels with the same intensity value, *t*, whose external boundary pixels all have a value greater than *t*.

By default, `imhmin` uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmin` uses `conndef(ndims(I),'maximal')`.

`I2 = imhmin(I,h,CONN)` computes the H-minima transform, where `CONN` specifies the connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
`4`	4-connected neighborhood
`8`	8-connected neighborhood
Three-dimensional connectivities	
`6`	6-connected neighborhood
`18`	18-connected neighborhood
`26`	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

`I` can be of any nonsparse numeric class and any dimension. `I2` has the same size and class as `I`.

imhmin

Example

```
a = 10*ones(10,10);
a(2:4,2:4) = 7;  % maxima 3 lower than surround
a(6:8,6:8) = 2;  % maxima 8 lower than surround
b = imhmin(a,4); % only the minima lower than 4 survive.
```

See Also `conndef`, `imhmax`, `imreconstruct`

Reference

[1] Pierre Soille, *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

Purpose

Impose minima

Syntax

```
I2 = imimposemin(I,BW)
```

Description

`I2 = imimposemin(I,BW)` modifies the intensity image `I` using morphological reconstruction so it only has regional minima wherever `BW` is nonzero. `BW` is a binary image the same size as `I`.

By default, `imimposemin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imimposemin` uses `conndef(ndims(I),'mimimum')`.

`I2 = imimposemin(I,H,CONN)` specifies the connectivity, where `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may also be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by-...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

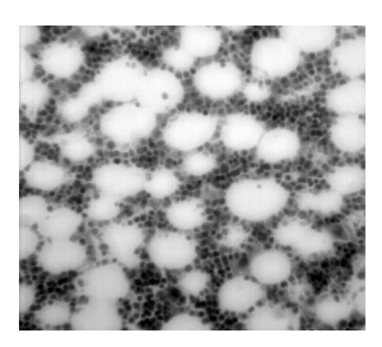
`I` can be of any nonsparse numeric class and any dimension. `BW` must be a nonsparse numeric array with the same size as `I`. `I2` has the same size and class as `I`.

Example

Modify the image in `bonemarr.tif` so that is only has regional minima at one location.

1 First read in and display the image to be processed, called the mask image.

```
I = imread('bonemarr.tif');
imshow(I)
```

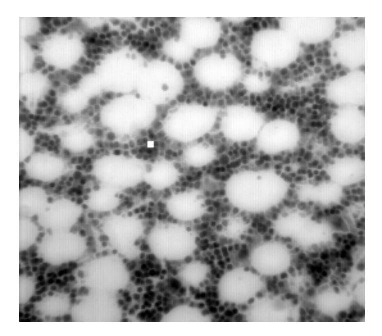


2 Create the marker image that will be used to process the mask image. The example creates a binary image the same size as the mask image, filled with zeros. It then sets a small area of the binary image to 1; these pixels define the location of the minima in the mask image.

```
bw = zeros(size(I));
bw(98:102,101:105) = 1;
```

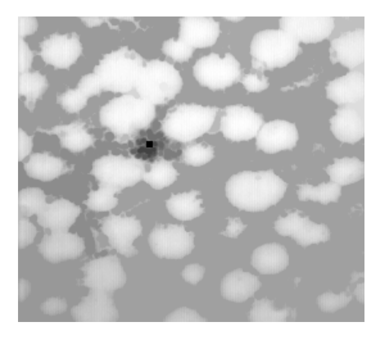
To show where these pixels of interest fall on the original image, this code superimposes the marker over the mask. The small white square marks the spot. This code is not essential to the impose minima operation.

```
J = I;
J(bw ~= 0) = 255;
figure, imshow(J)
```



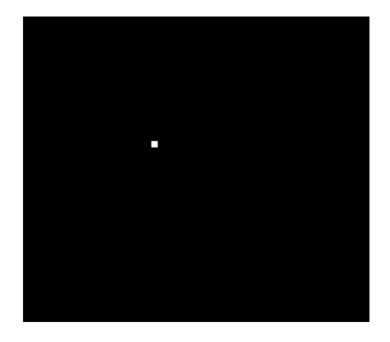
3 Impose the minina on the input image by performing morphological reconstruction of the mask image with the marker image. Note how all the dark areas of the original image, except those marked, are lighter.

```
K = imimposemin(I,bw);
figure, imshow(K)
```



To illustrate how this operation removes all minima in the original image except the imposed minima, the example uses `imregionalmin` to find all regional minima in the image.

```
bw2 = imregionalmin(K);
figure, imshow(bw2)
```



Algorithm imimposemin uses a technique based on morphological reconstruction.

See Also conndef, imreconstruct, imregionalmin

Purpose

Compute linear combination of images

Syntax

```
Z = imlincomb(K1,A1,K2,A2,...,Kn,An)
Z = imlincomb(K1,A1,K2,A2,...,Kn,An,K)
Z = imlincomb(..., output_class)
```

Description

`Z = imlincomb(K1,A1,K2,A2,...,Kn,An)` computes:

```
K1*A1 + K2*A2 + ... + Kn*An
```

where `K1`, `K2`, through `Kn` are real, double scalars and `A1`, `A2`, through `An` are real, nonsparse, numeric arrays with the same class and size. `Z` has the same class and size as `A1`.

`Z = imlincomb(K1,A1,K2,A2,...,Kn,An,K)` computes:

```
K1*A1 + K2*A2 + ... + Kn*An + K
```

where `imlincomb` adds `K`, a real, double scalar, to the sum of the products of `K1` through `Kn` and `A1` through `An`.

`Z = imlincomb(...,output_class)` lets you specify the class of `Z`. `output_class` is a string containing the name of a numeric class.

Note When performing a series of arithmetic operations on a pair of images, you can achieve more accurate results if you use `imlincomb` to combine the operations, rather than nesting calls to the individual arithmetic functions, such as `imadd`. When you nest calls to the arithmetic functions, and the input arrays are of an integer class, each function truncates and rounds the result before passing it to the next function, thus losing accuracy in the final result. `imlincomb` computes each element of the output, `Z`, individually in double-precision, floating point. If `Z` is an integer array, `imlincomb` truncates elements of `Z` that exceed the range of the integer type, and rounds off fractional values.

Example

Scale an image by a factor of two.

```
I = imread('cameraman.tif');
J = imlincomb(2,I);
```

```
imshow(J)
```

Form a difference image with the zero value shifted to 128.

```
I = imread('cameraman.tif');
J = uint8(filter2(fspecial('gaussian'), I));
K = imlincomb(1,I,-1,J,128); % K(r,c) = I(r,c) - J(r,c) + 128
imshow(K)
```

To illustrate how `imlincomb` performs all the arithmetic operations before truncating the result, compare the results of calculating the average of two arrays, `X` and `Y`, using nested arithmetic functions and then using `imlincomb`.

Consider the values in row 1, column 1: `255` in the `X` and `50` in `Y`. In the version that uses nested arithmetic functions, `imadd` adds `255` and `50` and truncates the result to `255` before passing it to `imdivide`. The average returned in `Z(1,1)` is `128`.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 20 50; 50 50 50 ]);
Z = imdivide(imadd(X,Y),2)
Z =
   128    15    63
    47   128    75
```

`imlincomb` performs the addition and division in double precision and only truncates the final result. The average returned in `Z2(1,1)` is `153`.

```
Z2 = imlincomb(.5,X,.5,Y)
Z2 =
   153    15    63
    47   138    75
```

Form a difference image with the zero value shifted to 128.

```
I = imread('cameraman.tif');
J = uint8(filter2(fspecial('gaussian'), I));
K = imlincomb(1,I,-1,J,128);
% K(r,c) = I(r,c) - J(r,c) + 128
imshow(K)
```

See Also

`imadd`, `imcomplement`, `imdivide`, `immultiply`, `imsubtract`

Purpose

Make a movie of a multiframe indexed image

Syntax

```
mov = immovie(X,map)
mov = immovie(RGB)
```

Description

mov = immovie(X,map) returns the movie structure array, mov, from the images in the multiframe indexed image X with the colormap map. As it creates the movie array, it displays the movie frames on the screen. You can play the movie using the MATLAB movie function. For details about the movie structure array, see the reference page for getframe.

X comprises multiple indexed images, all having the same size and all using the colormap map. X is an m-by-n-by-1-by-k array, where k is the number of images.

mov = immovie(RGB) returns the movie structure array mov from the images in the multiframe, truecolor image RGB.

RGB comprises multiple truecolor images, all having the same size. RGB is an m-by-n-by-3-by-k array, where k is the number of images.

Remarks

You can also use the MATLAB function avifile to make movies from images. The avifile function creates AVI files. In addition, you can convert an existing MATLAB movie into an AVI file by using the movie2avi function.

Class Support

An indexed image can be uint8, uint16, double, or logical. A truecolor image can be uint8, uint16, or double. mov is a MATLAB movie structure.

Example

```
load mri
mov = immovie(D,map);
movie(mov,3)
```

See Also

avifile, getframe, montage, movie, movie2avi

immultiply

Purpose Multiply two images, or multiply an image by a constant

Syntax

```
Z = immultiply(X,Y)
```

Description

Z = immultiply(X,Y) multiplies each element in array X by the corresponding element in array Y and returns the product in the corresponding element of the output array Z.

If X and Y are real numeric arrays with the same size and class, then Z has the same size and class as X. If X is a numeric array and Y is a scalar double, then Z has the same size and class as X.

If X is logical and Y is numeric, then Z has the same size and class as Y. If X is numeric and Y is logical, then Z has the same size and class as X.

immultiply computes each element of Z individually in double-precision floating point. If X is an integer array, then elements of Z exceeding the range of the integer type are truncated, and fractional values are rounded.

If X and Y are double arrays, you can use the expression X.*Y instead of this function.

Example

Multiply an image by itself. Note how the example converts the class of the image from uint8 to uint16 before performing the multiplication to avoid truncating the results.

```
I = imread('moon.tif');
I16 = uint16(I);
J = immultiply(I16,I16);
imshow(I), figure, imshow(J)
```

Scale an image by a constant factor:

```
I = imread('moon.tif');
J = immultiply(I,0.5);
subplot(1,2,1), imshow(I)
subplot(1,2,2), imshow(J)
```

See also

imabsdiff, imadd, imcomplement, imdivide, imlincomb, imsubtract

Purpose Add noise to an image

Syntax

```
J = imnoise(I,type)
J = imnoise(I,type,parameters)
```

Description

`J = imnoise(I,type)` adds noise of given type to the intensity image `I`. *type* is a string that can have one of these values:

- `'gaussian'` for Gaussian white noise
- `'localvar'` for zero-mean Gaussian white noise with an intensity-dependent variance
- `'poisson'` for Poisson noise
- `'salt & pepper'` for “on and off” pixels
- `'speckle'` for multiplicative noise

`J = imnoise(I,type,parameters)` accepts an algorithm *type* plus additional modifying parameters particular to the type of algorithm chosen. If you omit these arguments, `imnoise` uses default values for the parameters. Here are examples of the different noise types and their parameters:

- `J = imnoise(I,'gaussian',m,v)` adds Gaussian white noise of mean `m` and variance `v` to the image `I`. The default is zero mean noise with 0.01 variance.
- `J = imnoise(I,'localvar',V)` adds zero-mean, Gaussian white noise of local variance, `V`, to the image `I`. `V` is an array of the same size as `I`.
- `J = imnoise(I,'localvar',image_intensity,var)` adds zero-mean, Gaussian noise to an image `I`, where the local variance of the noise, `var`, is a function of the image intensity values in `I`. The `image_intensity` and `var` arguments are vectors of the same size, and `plot(image_intensity,var)` plots the functional releationship between noise variance and image intensity. The `image_intensity` vector must contain normalized intensity values ranging from 0 to 1.
- `J = imnoise(I,'poisson')` generates Poisson noise from the data instead of adding artificial noise to the data. In order to respect Poisson statistics, the intensities of `unit8` and `uint16` images must correspond to the number of photons (or any other quanta of information). Double-precision images are used when the number of photons per pixel can be much larger than 65535 (but less than 10^12); the intensities values vary between 0 and 1 and correspond to the number of photons divided by 10^12.

- `J = imnoise(I,'salt & pepper',d)` adds salt and pepper noise to the image I, where `d` is the noise density. This affects approximately `d*prod(size(I))` pixels. The default is 0.05 noise density.
- `J = imnoise(I,'speckle',v)` adds multiplicative noise to the image `I`, using the equation `J = I+n*I`, where `n` is uniformly distributed random noise with mean 0 and variance `v`. The default for `v` is 0.04.

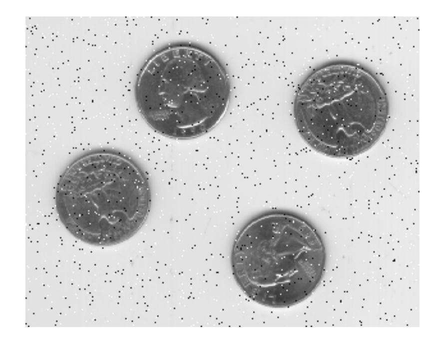
Class Support

`I` can be of class `uint8`, `uint16`, or `double`. The output image `J` is of the same class as `I`. If `I` has more than two dimensions it is treated as a multidimensional intensity image and not as an RGB image.

Example

```
I = imread('eight.tif');
J = imnoise(I,'salt & pepper',0.02);
imshow(I)
figure, imshow(J)
```





See Also

`rand`, `randn` in the MATLAB Function Reference

Purpose

Open an image

Syntax

```
IM2 = imopen(IM,SE)
IM2 = imopen(IM,NHOOD)
```

Description

`IM2 = imopen(IM,SE)` performs morphological opening on the grayscale or binary image `IM` with the structuring element `SE`. The argument `SE` must be a single structuring element object, as opposed to an array of objects.

`IM2 = imopen(IM,NHOOD)` performs opening with the structuring element `strel(NHOOD)`, where `NHOOD` is an array of 0's and 1's that specifies the structuring element neighborhood.

Class Support

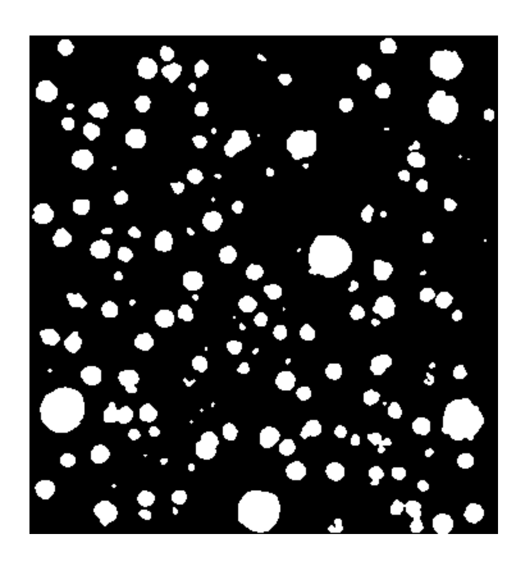
`IM` can be any numeric or logical class and any dimension, and must be nonsparse. If `IM` is logical, then `SE` must be flat. `IM2` has the same class as `IM`.

Example

This example uses `imopen` to filter out the smaller objects in an image.

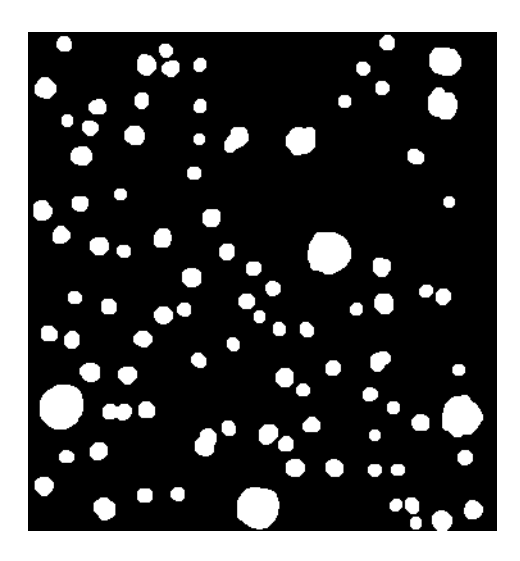
1 Read the image into the MATLAB workspace and threshold it.

```
I = imread('nodules1.tif');
bw = ~im2bw(I,graythresh(I));
imshow(bw), title('Thresholded Image')
```



2 Create a disk-shaped structuring element and open the image.

```
se = strel('disk',5);
bw2 = imopen(bw,se);
figure, imshow(bw2), title('After opening')
```



See Also

imclose, imdilate, imerode, strel

Purpose

Determine pixel color values

Syntax

```
P = impixel(I)
P = impixel(X,map)
P = impixel(RGB)

P = impixel(I,c,r)
P = impixel(X,map,c,r)
P = impixel(RGB,c,r)
[c,r,P] = impixel(...)

P = impixel(x,y,I,xi,yi)
P = impixel(x,y,X,map,xi,yi)
P = impixel(x,y,RGB,xi,yi)
[xi,yi,P] = impixel(x,y,...)
```

Description

`impixel` returns the red, green, and blue color values of specified image pixels. In the syntaxes below, `impixel` displays the input image and waits for you to specify the pixels with the mouse.

```
P = impixel(I)
P = impixel(X,map)
P = impixel(RGB)
```

If you omit the input arguments, `impixel` operates on the image in the current axes.

Use normal button clicks to select pixels. Press **Backspace** or **Delete** to remove the previously selected pixel. A shift-click, right-click, or double-click adds a final pixel and ends the selection; pressing **Return** finishes the selection without adding a pixel.

When you finish selecting pixels, `impixel` returns an `m`-by-3 matrix of RGB values in the supplied output argument. If you do not supply an output argument, `impixel` returns the matrix in `ans`.

You can also specify the pixels noninteractively, using these syntaxes.

```
P = impixel(I,c,r)
P = impixel(X,map,c,r)
P = impixel(RGB,c,r)
```

r and c are equal-length vectors specifying the coordinates of the pixels whose RGB values are returned in P. The kth row of P contains the RGB values for the pixel (r(k),c(k)).

If you supply three output arguments, impixel returns the coordinates of the selected pixels. For example,

```
[c,r,P] = impixel(...)
```

To specify a nondefault spatial coordinate system for the input image, use these syntaxes.

```
P = impixel(x,y,I,xi,yi)
P = impixel(x,y,X,map,xi,yi)
P = impixel(x,y,RGB,xi,yi)
```

x and y are two-element vectors specifying the image XData and YData. xi and yi are equal-length vectors specifying the spatial coordinates of the pixels whose RGB values are returned in P. If you supply three output arguments, impixel returns the coordinates of the selected pixels.

```
[xi,yi,P] = impixel(x,y,...)
```

Class Support

The input image can be of class uint8, uint16, double, or logical. All other inputs and outputs are of class double.

Remarks

impixel works with indexed, intensity, and RGB images. impixel always returns pixel values as RGB triplets, regardless of the image type:

- For an RGB image, impixel returns the actual data for the pixel. The values are either uint8 integers or double floating-point numbers, depending on the class of the image array.
- For an indexed image, impixel returns the RGB triplet stored in the row of the colormap that the pixel value points to. The values are double floating-point numbers.
- For an intensity image, impixel returns the intensity value as an RGB triplet, where R=G=B. The values are either uint8 integers or double floating-point numbers, depending on the class of the image array.

Example

```
RGB = imread('flowers.tif');
c = [12 146 410];
```

```
r = [104 156 129];
pixels = impixel(RGB,c,r)

pixels =

    61    59   101
   253   240     0
   237    37    44
```

See Also

`improfile`, `pixval`

improfile

Purpose

Compute pixel-value cross-sections along line segments

Syntax

```
c = improfile
c = improfile(n)

c = improfile(I,xi,yi)
c = improfile(I,xi,yi,n)

[cx,cy,c] = improfile(...)
[cx,cy,c,xi,yi] = improfile(...)

[...] = improfile(x,y,I,xi,yi)
[...] = improfile(x,y,I,xi,yi,n)

[...] = improfile(...,method)
```

Description

`improfile` computes the intensity values along a line or a multiline path in an image. `improfile` selects equally spaced points along the path you specify, and then uses interpolation to find the intensity value for each point. `improfile` works with grayscale intensity images and RGB images.

If you call `improfile` with one of these syntaxes, it operates interactively on the image in the current axes.

```
c = improfile
c = improfile(n)
```

`n` specifies the number of points to compute the intensity value for. If you do not provide this argument, `improfile` chooses a value for `n`, roughly equal to the number of pixels the path traverses.

You specify the line or path using the mouse, by clicking on points in the image. Press **Backspace** or **Delete** to remove the previously selected point. A shift-click, right-click, or double-click adds a final point and ends the selection; pressing **Return** finishes the selection without adding a point. When you finish selecting points, `improfile` returns the interpolated data values in `c`. `c` is an `n`-by-1 vector if the input is a grayscale intensity image, or an `n`-by-1-by-3 array if the input is an RGB image.

If you omit the output argument, `improfile` displays a plot of the computed intensity values. If the specified path consists of a single line segment, `improfile` creates a two-dimensional plot of intensity values versus the distance along the line segment; if the path consists of two or more line segments, `improfile` creates a three-dimensional plot of the intensity values versus their x- and y-coordinates.

You can also specify the path noninteractively, using these syntaxes.

```
c = improfile(I,xi,yi)
c = improfile(I,xi,yi,n)
```

`xi` and `yi` are equal-length vectors specifying the spatial coordinates of the endpoints of the line segments.

You can use these syntaxes to return additional information.

```
[cx,cy,c] = improfile(...)
[cx,cy,c,xi,yi] = improfile(...)
```

`cx` and `cy` are vectors of length `n`, containing the spatial coordinates of the points at which the intensity values are computed.

To specify a nondefault spatial coordinate system for the input image, use these syntaxes.

```
[...] = improfile(x,y,I,xi,yi)
[...] = improfile(x,y,I,xi,yi,n)
```

`x` and `y` are two-element vectors specifying the image `XData` and `YData`.

`[...] = improfile(...,method)` uses the specified interpolation method. `method` is a string that can have one of these values:

- `'nearest'` (default) uses nearest neighbor interpolation.
- `'bilinear'` uses bilinear interpolation.
- `'bicubic'` uses bicubic interpolation.

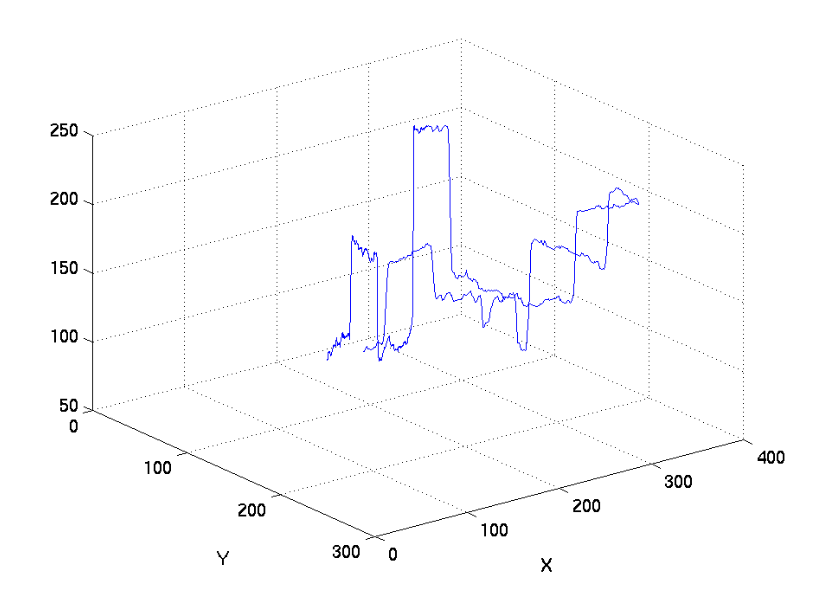
If you omit the `method` argument, `improfile` uses the default method of `'nearest'`.

Class Support

The input image can be `uint8`, `uint16`, `double`, or `logical`. All other inputs and outputs must be `double`.

Example

```
I = imread('alumgrns.tif');
x = [35 338 346 103];
y = [253 250 17 148];
improfile(I,x,y), grid on
```



See Also

impixel, pixval

interp2 in the MATLAB Function Reference

Purpose

Read image from graphics file

`imread` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose Morphological reconstruction

Syntax

```
IM = imreconstruct(MARKER,MASK)
IM = imreconstruct(MARKER,MASK,CONN)
```

Description

`IM = imreconstruct(MARKER,MASK)` performs morphological reconstruction of the image `MARKER` under the image `MASK`. `MARKER` and `MASK` can be two intensity images or two binary images with the same size. The returned image, `IM` is an intensity or binary image, respectively. `MARKER` must be the same size as `MASK`, and its elements must be less than or equal to the corresponding elements of `MASK`.

By default, `imreconstruct` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imreconstruct` uses `conndef(ndims(I),'maximal')`.

`IM = imreconstruct(MARKER,MASK,CONN)` performs morphological reconstruction with the specified connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ... -by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Morphological reconstruction is the algorithmic basis for several other Image Processing Toolbox functions, including `imclearborder`, `imextendedmax`, `imextendedmin`, `imfill`, `imhmax`, `imhmin`, and `imimposemin`.

Class Support

`MARKER` and `MASK` must be nonsparse numeric or logical arrays with the same class and any dimension. `IM` is of the same class as `MARKER` and `MASK`.

Algorithm

`imreconstruct` uses the "fast hybrid grayscale reconstruction" algorithm described in [1].

See Also

`imclearborder`, `imextendedmax`, `imextendedmin`, `imfill`, `imhmax`, `imhmin`, `imimposemin`

Reference

[1] Luc Vincent, "Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms," *IEEE Transactions on Image Processing*, vol. 2, no. 2, April 1993, pp. 176-201.

imregionalmax

Purpose

Find regional maxima

Syntax

```
BW = imregionalmax(I)
BW = imregionalmax(I,CONN)
```

Description

`BW = imregionalmax(I)` finds the regional maxima of `I`. `imregionalmax` returns a binary image, `BW`, the same size as `I`, that identifies the locations of the regional maxima in `I`. In `BW`, pixels that are set to `1` identify regional maxima; all other pixels are set to 0.

Regional maxima are connected components of pixels with the same intensity value, *t*, whose external boundary pixels all have a value less than *t*.

By default, `imregionalmax` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imregionalmax` uses `conndef(ndims(I),'maximal')`.

`BW = imregionalmax(I,CONN)` computes the regional maxima of `I` using the specified connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

I can be any nonsparse, numeric class and any dimension. BW is logical.

Example

```
A = 10*ones(10,10);
A(2:4,2:4) = 22;  % maxima 12 higher than surround
A(6:8,6:8) = 33;  % maxima 23 higher than surround
A(2,7) = 44;
A(3,8) = 45;
A(4,9) = 44;
A =
    10    10    10    10    10    10    10    10    10    10
    10    22    22    22    10    10    44    10    10    10
    10    22    22    22    10    10    10    45    10    10
    10    22    22    22    10    10    10    10    44    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    33    33    33    10    10
    10    10    10    10    10    33    33    33    10    10
    10    10    10    10    10    33    33    33    10    10
    10    10    10    10    10    10    10    10    10    10
    10    10    10    10    10    10    10    10    10    10

regmax = imregionalmax(A)
regmax =
     0     0     0     0     0     0     0     0     0     0
     0     1     1     1     0     0     0     0     0     0
     0     1     1     1     0     0     0     1     0     0
     0     1     1     1     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

See Also

conndef, imreconstruct, imregionalmin

imregionalmin

Purpose

Find regional minima

Syntax

```
BW = imregionalmin(I)
BW = imregionalmin(I,CONN)
```

Description

`BW = imregionalmin(I)` computes the regional minima of `I`. The output binary image `BW` has value 1 corresponding to the pixels of `I` that belong to regional minima and `0` otherwise. `BW` is the same size as `I`.

Regional minima are connected components of pixels with the same intensity value, t, whose external boundary pixels all have a value greater than t.

By default, `imregionalmin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imregionalmin` uses `conndef(ndims(I),'maximal')`.

`BW = imregionalmin(I,CONN)` specifies the desired connectivity. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

`I` can be any nonsparse, numeric class and any dimension. `BW` is `logical`.

Example

```
A = 10*ones(10,10);
A(2:4,2:4) = 3;  % minima 3 lower than surround
A(6:8,6:8) = 8;  % minima 8 lower than surround
A =
   10    10    10    10    10    10    10    10    10    10
   10     7     7     7    10    10    10    10    10    10
   10     7     7     7    10    10    10    10    10    10
   10     7     7     7    10    10    10    10    10    10
   10    10    10    10    10    10    10    10    10    10
   10    10    10    10    10     2     2     2    10    10
   10    10    10    10    10     2     2     2    10    10
   10    10    10    10    10     2     2     2    10    10
   10    10    10    10    10    10    10    10    10    10
   10    10    10    10    10    10    10    10    10    10

B = imregionalmin(A)
B =
      0     0     0     0     0     0     0     0     0     0
      0     1     1     1     0     0     0     0     0     0
      0     1     1     1     0     0     0     0     0     0
      0     1     1     1     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     1     1     1     0     0
      0     0     0     0     0     1     1     1     0     0
      0     0     0     0     0     1     1     1     0     0
      0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0
```

See Also

`conndef`, `imreconstruct`, `imregionalmax`

imresize

Purpose Resize an image

Syntax

```
B = imresize(A,m,method)
B = imresize(A,[mrows ncols],method)

B = imresize(...,method,n)
B = imresize(...,method,h)
```

Description

`imresize` resizes an image of any type using the specified interpolation method. *`method`* is a string that can have one of these values:

- `'nearest'` (default) uses nearest neighbor interpolation.
- `'bilinear'` uses bilinear interpolation.
- `'bicubic'` uses bicubic interpolation.

If you omit the *`method`* argument, `imresize` uses the default method of `'nearest'`.

`B = imresize(A,m,method)` returns an image that is `m` times the size of `A`. If `m` is between 0 and 1.0, `B` is smaller than `A`. If `m` is greater than 1.0, `B` is larger than `A`.

`B = imresize(A,[mrows ncols],method)` returns an image of size `[mrows ncols]`. If the specified size does not produce the same aspect ratio as the input image has, the output image is distorted.

When the specified output size is smaller than the size of the input image, and `method` is `'bilinear'` or `'bicubic'`, `imresize` applies a lowpass filter before interpolation to reduce aliasing. The default filter size is 11-by-11.

You can specify a different order for the default filter using

```
[...] = imresize(...,method,n)
```

`n` is an integer scalar specifying the size of the filter, which is `n`-by-`n`. If `n` is 0 (zero), `imresize` omits the filtering step.

You can also specify your own filter `h` using

```
[...] = imresize(...,method,h)
```

`h` is any two-dimensional FIR filter (such as those returned by `ftrans2`, `fwind1`, `fwind2`, or `fsamp2`).

Class Support

The input image, `A`, can be numeric or logical and it must be nonsparse. The output image, `B`, is of the same class as the input image.

See Also

`imrotate`, `imtransform`, `tformarray`; `interp2` in the MATLAB Function Reference

imrotate

Purpose Rotate an image

Syntax

```
B = imrotate(A,angle,method)
B = imrotate(A,angle,method,'crop')
```

Description `B = imrotate(A,angle,method)` rotates the image `A` by `angle` degrees in a counter-clockwise direction, using the specified interpolation method. *`method`* is a string that can have one of these values:

- `'nearest'` (default) uses nearest neighbor interpolation.
- `'bilinear'` uses bilinear interpolation.
- `'bicubic'` uses bicubic interpolation.

If you omit the *`method`* argument, `imrotate` uses the default method of `'nearest'`.

The returned image matrix `B` is, in general, larger than `A` to include the whole rotated image. `imrotate` sets invalid values on the periphery of `B` to 0.

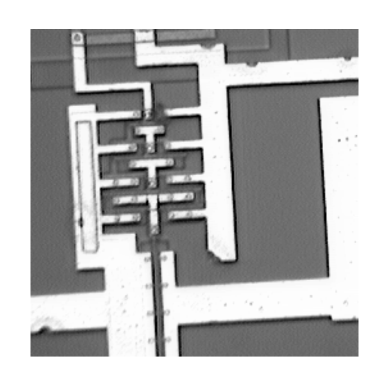
`B = imrotate(A,angle,method,'crop')` rotates the image `A` through `angle` degrees and returns the central portion which is the same size as `A`.

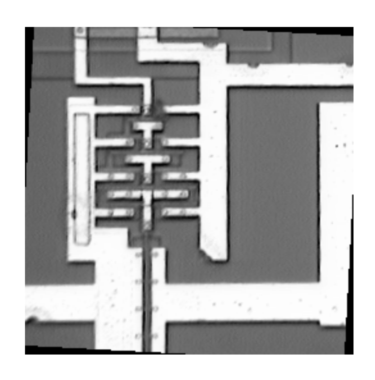
Class Support The input image, `A`, can be numeric or logical and it must be nonsparse. The output image, `B`, is of the same class as the input image.

Remarks To rotate the image clockwise, specify a negative angle.

Example

```
I = imread('ic.tif');
J = imrotate(I,-4,'bilinear','crop');
imshow(I)
figure, imshow(J)
```





See Also imcrop, imresize, imtransform, tformarray

imshow

Purpose Display an image

Syntax

```
imshow(I,n)
imshow(I,[low high])
imshow(BW)
imshow(X,map)
imshow(RGB)
imshow(...,display_option)

imshow(x,y,A,...)
imshow filename
h = imshow(...)
```

Description

`imshow(I,n)` displays the intensity image `I` with `n` discrete levels of gray. If you omit `n`, `imshow` uses 256 gray levels on 24-bit displays, or 64 gray levels on other systems.

`imshow(I,[low high])` displays `I` as a grayscale intensity image, specifying the data range for `I`. The value `low` (and any value less than `low`) displays as black, the value `high` (and any value greater than `high`) displays as white, and values in between display as intermediate shades of gray. `imshow` uses the default number of gray levels. If you use an empty matrix (`[]`) for `[low high]`, `imshow` uses `[min(I(:)) max(I(:))]`; the minimum value in `I` displays as black, and the maximum value displays as white.

`imshow(BW)` displays the binary image `BW`. Values of 0 display as black, and values of 1 display as white.

`imshow(X,map)` displays the indexed image `X` with the colormap `map`.

`imshow(RGB)` displays the truecolor image `RGB`.

`imshow(...,display_option)` displays the image, calling `truesize` if `display_option` is `'truesize'`, or suppressing the call to `truesize` if `display_option` is `'notruesize'`. Either option string can be abbreviated. If you do not supply this argument, `imshow` determines whether to call `truesize` based on the setting of the `'ImshowTruesize'` preference.

`imshow(x,y,A,...)` uses the two-element vectors `x` and `y` to establish a nondefault spatial coordinate system, by specifying the image `XData` and `YData`.

imshow filename displays the image stored in the graphics file filename. imshow calls imread to read the image from the file, but the image data is not stored in the MATLAB workspace. The file must be in the current directory or on the MATLAB path.

h = imshow(...) returns the handle to the image object created by imshow.

Class Support

The input image can be of class logical, uint8, uint16, or double, and it must be nonsparse.

Remarks

You can use the iptsetpref function to set several toolbox preferences that modify the behavior of imshow. For example:

- 'ImshowBorder' controls whether imshow displays the image with a border around it.
- 'ImshowAxesVisible' controls whether imshow displays the image with the axes box and tick labels.
- 'ImshowTruesize' controls whether imshow calls the truesize function.

Note that the display_option argument to imshow enables you to override the 'ImshowTruesize' preference.

For more information about these preferences, see the reference entry for iptsetpref.

See Also

getimage, imread, iptgetpref, iptsetpref, subimage, truesize, warp

image, imagesc in the MATLAB Function Reference

imsubtract

Purpose Subtract one image from another, or subtract a constant from an image

Syntax

```
Z = imsubtract(X,Y)
```

Description `Z = imsubtract(X,Y)` subtracts each element in array `Y` from the corresponding element in array `X` and returns the difference in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays of the same size and class, or `Y` is a double scalar. The array returned, `Z`, has the same size and class as `X`.

If `X` is an integer array, then elements of the output that exceed the range of the integer type are truncated, and fractional values are rounded.

If `X` and `Y` are double arrays, then you can use the expression `X-Y` instead of this function.

Examples Subtract two `uint8` arrays. Note that negative results are rounded to `0`.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imadd(X,Y)
Z =

   205    0   25
     0  175   50
```

Estimate and subtract the background of the rice image:

```
I = imread('rice.tif');
blocks = blkproc(I,[32 32],'min(x(:))');
background = imresize(blocks,[256 256],'bilinear');
Ip = imsubtract(I,background);
imshow(Ip,[])
```

Subtract a constant value from the rice image:

```
I = imread('rice.tif');
Iq = imsubtract(I,50);
subplot(1,2,1), imshow(I)
subplot(1,2,2), imshow(Iq)
```

See also `imabsdiff`, `imadd`, `imcomplement`, `imdivide`, `imlincomb`, `immultiply`

Purpose Perform top-hat filtering

Syntax

```
IM2 = imtophat(IM,SE)
IM2 = imtophat(IM,NHOOD)
```

Description `IM2 = imtophat(IM,SE)` performs morphological top-hat filtering on the grayscale or binary input image `IM` using the structuring element `SE`, where `SE` is returned by `strel`. `SE` must be a single structuring element object, not an array containing multiple structuring element objects.

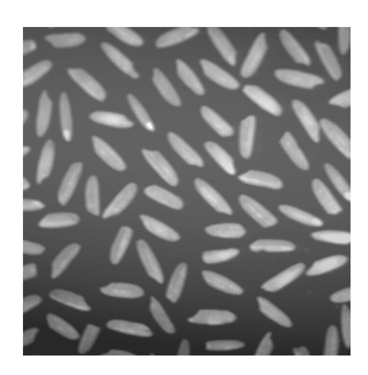
`IM2 = imtophat(IM,NHOOD)`, where `NHOOD` is an array of 0's and 1's that specifies the size and shape of the structuring element, is the same as `imptophat(IM,strel(NHOOD))`.

Class Support `IM` can be numeric or logical and must be nonsparse. The output image, `IM2`, has the same class as the input image. If the input is binary (logical), then the structuring element must be flat.

Example You can use top-hat filtering to correct uneven illumination when the background is dark. This example uses top-hat filtering with a disk-shaped structuring element to remove the uneven background illumination from the image `rice.tif`.

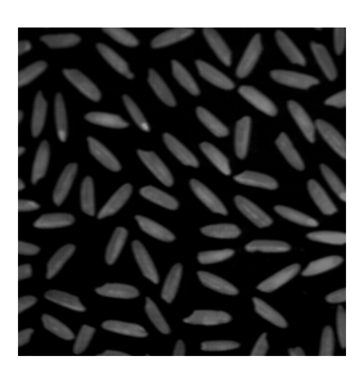
1 Read the image into the MATLAB workspace.

```
I = imread('rice.tif');
imshow(I), title('Original')
```



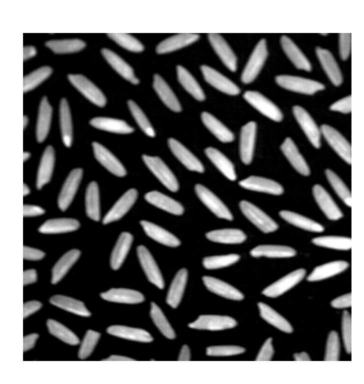
2 Create the structuring element and perform top-hat filtering of the image.

```
se = strel('disk',12);
J = imtophat(I,se);
figure, imshow(J), title('Step 1: Top-hat filtering')
```



3 Use `imadjust` and `stretchlim` to make the result more easily visible.

```
K = imadjust(J,stretchlim(J));
figure, imshow(K), title('Step 2: Contrast adjustment')
```



See Also

`imbothat`, `strel`

Purpose

Apply 2-D spatial transformation to image

Syntax

```
B = imtransform(A,TFORM)
B = imtransform(A,TFORM,INTERP)
[B,XDATA,YDATA] = imtransform(...)
[B,XDATA,YDATA] = imtransform(...,PARAM1,VAL1,PARAM2,VAL2,...)
```

Description

`B = imtransform(A,TFORM)` transforms the image `A` according to the 2-D spatial transformation defined by `TFORM`, which is a spatial transformation structure (`TFORM`) as returned by `maketform` or `cp2tform`. If `ndims(A) > 2`, such as for an RGB image, then the same 2-D transformation is automatically applied to all 2-D planes along the higher dimensions.

When you use this syntax, `imtransform` automatically shifts the origin of your output image to make as much of the transformed image visible as possible. If you are using `imtransform` to do image registration, this syntax is not likely to give you the results you expect; you may want to set `'XData'` and `'YData'` explicitly. For more information, see Parameters, as well as Example 3.

`B = imtransform(A,TFORM,INTERP)` specifies the form of interpolation to use. `INTERP` can be one of the strings `'nearest'`, `'bilinear'`, or `'bicubic'`. Alternatively, `INTERP` can be a `RESAMPLER` structure as returned by `makeresampler`. This option allows more control over how resampling is performed. The default value for `INTERP` is `'bilinear'`.

`[B,XDATA,YDATA] = imtransform(...)` returns the location of the output image `B` in the output X-Y space. `XDATA` and `YDATA` are two-element vectors. The elements of `XDATA` specify the x-coordinates of the first and last columns of `B`. The elements of `YDATA` specify the y-coordinates of the first and last rows of `B`. Normally, `imtransform` computes `XDATA` and `YDATA` automatically so that B contains the entire transformed image `A`. However, you can override this automatic computation; see below.

`[B,XDATA,YDATA] = imtransform(...,PARAM1,VAL1,PARAM2,VAL2,...)` specifies parameters that control various aspects of the spatial transformation. Parameter names can be abbreviated, and case does not matter.

Parameters

This table lists all the parameters you can specify. Note that parameter names can be abbreviated and are not case-sensitive.

Parameter	Description
`'UData'` `'VData'`	Both of these parameters are two-element real vectors. `'UData'` and `'VData'` specify the spatial location of the image `A` in the 2-D input space, U-V. The two elements of `'UData'` give the *u*-coordinates (horizontal) of the first and last columns of `A`, respectively. The two elements of `'VData'` give the *v*-coordinates (vertical) of the first and last rows of `A`, respectively. The default values for `'UData'` and `'VData'` are `[1 size(A,2)]` and `[1 size(A,1)]`, respectively.
`'XData'` `'YData'`	Both of these parameters are two-element real vectors. `'XData'` and `'YData'` specify the spatial location of the output image `B` in the 2-D output space X-Y. The two elements of `'XData'` give the *x*-coordinates (horizontal) of the first and last columns of `B`, respectively. The two elements of `'YData'` give the *y*-coordinates (vertical) of the first and last rows of `B`, respectively.
	If `'XData'` and `'YData'` are not specified, `imtransform` estimates values for them that will completely contain the entire transformed output image.

Parameter	Description
'XYScale'	A one- or two-element real vector. The first element of 'XYScale' specifies the width of each output pixel in X-Y space. The second element (if present) specifies the height of each output pixel. If 'XYScale' has only one element, then the same value is used for both width and height. If 'XYScale' is not specified but 'Size' is, then 'XYScale' is computed from 'Size', 'XData', and 'YData'. If neither 'XYScale' nor 'Size' is provided, then the scale of the input pixels is used for 'XYScale'.
'Size'	A two-element vector of nonnegative integers. 'Size' specifies the number of rows and columns of the output image B. For higher dimensions, the size of B is taken directly from the size of A. In other words, size(B,k) equals size(A,k) for k > 2. If 'Size' is not specified, then it is computed from 'XData', 'YData', and 'XYScale'.

Parameter	Description
'FillValues'	An array containing one or several fill values. Fill values are used for output pixels when the corresponding transformed location in the input image is completely outside the input image boundaries. If A is 2-D, 'FillValues' must be a scalar. However, if A's dimension is greater than two, then 'FillValues' can be an array whose size satisfies the following constraint: size(fill_values,k) must either equal size(A,k+2) or 1. For example, if A is a uint8 RGB image that is 200-by-200-by-3, then possibilities for 'FillValues' include: 0 - fill with black [0;0;0] - also fill with black 255 - fill with white [255;255;255] - also fill with white [0;0;255] - fill with blue [255;255;0] - fill with yellow
	If A is 4-D with size 200-by-200-by-3-by-10, then 'FillValues' can be a scalar, 1-by-10, 3-by-1, or 3-by-10.

Notes

- When you do not specify the output-space location for B using 'XData' and 'YData', imtransform estimates them automatically using the function findbounds. For some commonly-used transformations, such as affine or projective, for which a forward-mapping is easily computable, findbounds is fast. For transformations that do not have a forward mapping, such as the polynomial ones computed by cp2tform, findbounds can take significantly longer. If you can specify 'XData' and 'YData' directly for such transformations, imtransform may run noticeably faster.
- The automatic estimate of 'XData' and 'YData' using findbounds is not guaranteed in all cases to completely contain all the pixels of the transformed input image.

- The output values XDATA and YDATA may not exactly equal the input 'XData and 'YData' parameters. This can happen either because of the need for an integer number or rows and columns, or if you specify values for 'XData', 'YData', 'XYScale', and 'Size' that are not entirely consistent. In either case, the first element of XDATA and YDATA always equals the first element of 'XData' and 'YData', respectively. Only the second elements of XDATA and YDATA might be different.
- imtransform assumes spatial-coordinate conventions for the transformation TFORM. Specifically, the first dimension of the transformation is the horizontal or *x*-coordinate, and the second dimension is the vertical or *y*-coordinate. Note that this is the reverse of the array subscripting convention in MATLAB.
- TFORM must be a 2-D transformation to be used with imtransform. For arbitrary-dimensional array transformations, see tformarray.

Class Support

The input image, A, can be of any nonsparse numeric class, real or complex, or it can be of class logical. The class of B is the same as the class of A.

Example

Example 1

Apply a horizontal shear to an intensity image.

```
I = imread('cameraman.tif');
tform = maketform('affine',[1 0 0; .5 1 0; 0 0 1]);
J = imtransform(I,tform);
imshow(I), figure, imshow(J)
```

Example 2

A projective transformation can map a square to a quadrilateral. In this example, set up an input coordinate system so that the input image fills the unit square and then transform the image into the quadrilateral with vertices (0 0), (1 0), (1 1), (0 1) to the quadrilateral with vertices (-4 2), (-8 3), (-3 -5), and (6 3). Fill with gray and use bicubic interpolation. Make the output size the same as the input size.

```
I = imread('cameraman.tif');
udata = [0 1];  vdata = [0 1];  % input coordinate system
tform = maketform('projective',[ 0 0;  1  0;  1  1; 0 1],...
                               [-4 2; -8 -3; -3 -5; 6 3]);
```

```
[B,xdata,ydata] =
imtransform(I,tform,'bicubic','udata',udata,...
                              'vdata',vdata,...
                              'size',size(I),...
                              'fill',128);
subplot(1,2,1), imshow(udata,vdata,I), axis on
subplot(1,2,2), imshow(xdata,ydata,B), axis on
```

Example 3

Register an aerial photo to an orthophoto.

```
unregistered = imread('westconcordaerial.png');
figure, imshow(unregistered)
figure, imshow('westconcordorthophoto.png')
load westconcordpoints % load some points that were already picked
t_concord = cp2tform(input_points,base_points,'projective');
info = imfinfo('westconcordorthophoto.png');
registered = imtransform(unregistered,t_concord,...
    'XData',[1 info.Width], 'YData',[1 info.Height]);
figure, imshow(registered)
```

See Also

cp2tform, imresize, imrotate, maketform, makeresampler, tformarray

Purpose

Write image to graphics file

`imwrite` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

ind2gray

Purpose Convert an indexed image to an intensity image

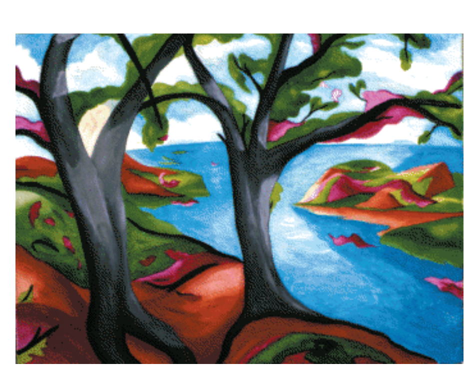
Syntax `I = ind2gray(X,map)`

Description `I = ind2gray(X,map)` converts the image `X` with colormap `map` to an intensity image `I`. `ind2gray` removes the hue and saturation information from the input image while retaining the luminance.

Class Support `X` can be of class `uint8`, `uint16`, or `double`. `I` is of class `double`.

Example

```
load trees
I = ind2gray(X,map);
imshow(X,map)
figure,imshow(I)
```





Algorithm `ind2gray` converts the colormap to NTSC coordinates using `rgb2ntsc`, and sets the hue and saturation components (*I* and *Q*) to zero, creating a gray colormap. `ind2gray` then replaces the indices in the image `X` with the corresponding grayscale intensity values in the gray colormap.

See Also `gray2ind`, `imshow`, `rgb2ntsc`

Purpose Convert an indexed image to an RGB image

Syntax

```
RGB = ind2rgb(X,map)
```

Description `RGB = ind2rgb(X,map)` converts the matrix `X` and corresponding colormap `map` to RGB (truecolor) format.

Class Support `X` can be of class `uint8`, `uint16`, or `double`. `RGB` is an `m`-by-`n`-by-3 array of class `double`.

See Also `ind2gray`, `rgb2ind`

iptgetpref

Purpose Get Image Processing Toolbox preference

Syntax

```
value = iptgetpref(prefname)
```

Description

`value = iptgetpref(prefname)` returns the value of the Image Processing Toolbox preference specified by the string `prefname`. For Preferences for a complete list of valid preference names. Preference names are not case-sensitive and can be abbreviated.

`iptgetpref` without an input argument displays the current setting of all Image Processing Toolbox preferences.

Example

```
value = iptgetpref('ImshowAxesVisible')

value =

off
```

See Also

`imshow`, `iptsetpref`

Purpose Set Image Processing Toolbox preferences or display valid values

Syntax

```
iptsetpref(prefname,value)
```

Description `iptsetpref(prefname)` displays the valid values for `prefname`. See Preferences for a complete list of available preferences.

`iptsetpref(prefname,value)` sets the Image Processing Toolbox preference specified by the string `prefname` to `value`. The setting persists until the end of the current MATLAB session, or until you change the setting. (To make the value persist between sessions, put the command in your `startup.m` file.)

Preferences This table describes the available preferences. Note that the preference names are case insensitive and can be abbreviated.

Preference Name	Values	Description
`'ImshowBorder'`	`'loose'` (default) or `'tight'`	If `'ImshowBorder'` is `'loose'`, `imshow` displays the image with a border between the image and the edges of the figure window, thus leaving room for axes labels, titles, etc. If `'ImshowBorder'` is `'tight'`, `imshow` adjusts the figure size so that the image entirely fills the figure. Note: There may still be a border if the image is very small, or if there are other objects besides the image and its axes in the figure.
`'ImshowAxesVisible'`	`'on'` or `'off'` (default)	If `'ImshowAxesVisible'` is `'on'`, `imshow` displays the image with the axes box and tick labels. If `'ImshowAxesVisible'` is `'off'`, `imshow` displays the image without the axes box and tick labels.

Preference Name	Values	Description
'ImshowTruesize'	'auto' (default) or 'manual'	If 'ImshowTruesize' is 'manual', imshow does not call truesize. If 'ImshowTruesize' is 'auto', imshow automatically decides whether to call truesize. (imshow calls truesize if there will be no other objects in the resulting figure besides the image and its axes.) You can override this setting for an individual display by specifying the display_option argument to imshow, or you can call truesize manually after displaying the image.
'TruesizeWarning'	'on' (default) or 'off'	If 'TruesizeWarning' is 'on', the truesize function displays a warning if the image is too large to fit on the screen. (The entire image is still displayed, but at less than true size.) If 'TruesizeWarning' is 'off', truesize does not display the warning. Note: This preference applies even when you call truesize indirectly, such as through imshow.

Example

```
iptsetpref('ImshowBorder','tight')
```

See Also

imshow, iptgetpref, truesize

axis in the MATLAB Function Reference

iradon

Purpose Compute inverse Radon transform

Syntax

```
I = iradon(P,theta)
I = iradon(P,theta,interp,filter,d,n)
[I,h] = iradon(...)
```

Description

`I = iradon(P,theta)` reconstructs the image `I` from projection data in the two-dimensional array `P`. The columns of `P` are parallel beam projection data. `iradon` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(P,1)/2)`.

`theta` describes the angles (in degrees) at which the projections were taken. It can be either a vector containing the angles or a scalar specifying `D_theta`, the incremental angle between projections. If `theta` is a vector, it must contain angles with equal spacing between them. If `theta` is a scalar specifying `D_theta`, the projections are taken at angles `theta = m*D_theta`, where `m = 0,1,2,...,size(P,2) 1`. If the input is the empty matrix (`[]`), `D_theta` defaults to `180/size(P,2)`.

`iradon` uses the filtered back-projection algorithm to perform the inverse Radon transform. The filter is designed directly in the frequency domain and then multiplied by the FFT of the projections. The projections are zero-padded to a power of 2 before filtering to prevent spatial domain aliasing and to speed up the FFT.

`I = iradon(P,theta,`*`interp`*`,`*`filter`*`,d,n)` specifies parameters to use in the inverse Radon transform. You can specify any combination of the last four arguments. `iradon` uses default values for any of these arguments that you omit.

`interp` specifies the type of interpolation to use in the backprojection. The available options are listed in order of increasing accuracy and computational complexity:

- `'nearest'` – nearest neighbor interpolation
- `'linear'` – linear interpolation (default)
- `'spline'` – spline interpolation

`filter` specifies the filter to use for frequency domain filtering. *`filter`* is a string that specifies any of the following standard filters:

- 'Ram-Lak' – The cropped Ram-Lak or ramp filter (default). The frequency response of this filter is | f |. Because this filter is sensitive to noise in the projections, one of the filters listed below may be preferable. These filters multiply the Ram-Lak filter by a window that de-emphasizes high frequencies.
- 'Shepp-Logan' – The Shepp-Logan filter multiplies the Ram-Lak filter by a sinc function.
- 'Cosine' – The cosine filter multiplies the Ram-Lak filter by a cosine function.
- 'Hamming' – The Hamming filter multiplies the Ram-Lak filter by a Hamming window.
- 'Hann' – The Hann filter multiplies the Ram-Lak filter by a Hann window.

d is a scalar in the range (0,1] that modifies the filter by rescaling its frequency axis. The default is 1. If d is less than 1, the filter is compressed to fit into the frequency range [0,d], in normalized frequencies; all frequencies above d are set to 0.

n is a scalar that specifies the number of rows and columns in the reconstructed image. If n is not specified, the size is determined from the length of the projections.

```
n = 2*floor(size(P,1)/(2*sqrt(2)))
```

If you specify n, iradon reconstructs a smaller or larger portion of the image, but does not change the scaling of the data. If the projections were calculated with the radon function, the reconstructed image may not be the same size as the original image.

[I,h] = iradon(...) returns the frequency response of the filter in the vector h.

Class Support

All input arguments and output arguments must be of class double.

Example

```
P = phantom(128);
R = radon(P,0:179);
I = iradon(R,0:179,'nearest','Hann');
imshow(P)
figure, imshow(I)
```





Algorithm

`iradon` uses the filtered backprojection algorithm to perform the inverse Radon transform. The filter is designed directly in the frequency domain and then multiplied by the FFT of the projections. The projections are zero-padded to a power of 2 before filtering to prevent spatial domain aliasing and to speed up the FFT.

See Also

`radon`, `phantom`

References

[1] Kak, Avinash C., and Malcolm Slaney, *Principles of Computerized Tomographic Imaging.* New York: IEEE Press.

isbw

Purpose Return true for a binary image

Syntax

```
flag = isbw(A)
```

Description

`flag = isbw(A)` returns 1 if `A` is a binary image and 0 otherwise.

The input image, `A`, is considered to be a binary image if it is a nonsparse logical array.

Class Support The input image, `A`, can be any MATLAB array.

See Also `isind`, `isgray`, `isrgb`

Purpose Return true for flat structuring element

Syntax

```
TF = isflat(SE)
```

Description `TF = isflat(SE)` returns true (`1`) if the structuring element `SE` is flat; otherwise it returns false (`0`). If `SE` is an array of `STREL` objects, then `TF` is the same size as `SE`.

Class Support `SE` is a `STREL` object. `TF` is a double precision value.

See Also `strel`

isgray

Purpose Return true for intensity image

Syntax

```
flag = isgray(A)
```

Description `flag = isgray(A)` returns 1 if `A` is a grayscale intensity image and 0 otherwise.

`isgray` uses these criteria to decide if `A` is an intensity image:

- If `A` is of class `double`, all values must be in the range [0,1], and the number of dimensions of `A` must be 2.
- If `A` is of class `uint16` or `uint8`, the number of dimensions of `A` must be 2.

Note A four-dimensional array that contains multiple intensity images returns 0, not 1.

Class Support The input image, `A`, can be of class `logical`, `uint8`, `uint16`, or `double`.

See Also `isbw`, `isind`, `isrgb`

isind

Purpose Return true for an indexed image

Syntax

```
flag = isind(A)
```

Description `flag = isind(A)` returns 1 if `A` is an indexed image and 0 otherwise.

`isind` uses these criteria to determine if `A` is an indexed image:

- If `A` is of class `double`, all values in `A` must be integers greater than or equal to 1, and the number of dimensions of `A` must be 2.
- If `A` is of class `uint8` or `uint16`, the number of dimensions of `A` must be 2.

Note A four-dimensional array that contains multiple indexed images returns 0, not 1.

Class Support `A` can be of class `logical`, `uint8`, `uint16`, or `double`.

See Also `isbw`, `isgray`, `isrgb`

isrgb

Purpose Return true for an RGB image

Syntax

```
flag = isrgb(A)
```

Description

`flag = isrgb(A)` returns 1 if `A` is an RGB truecolor image and 0 otherwise.

`isrgb` uses these criteria to determine if `A` is an RGB image:

- If `A` is of class `double`, all values must be in the range [0,1], and `A` must be `m`-by-`n`-by-3.
- If `A` is of class `uint16` or `uint8`, `A` must be `m`-by-`n`-by-3.

Note A four-dimensional array that contains multiple RGB images returns 0, not 1.

Class Support `A` can be of class `logical`, `uint8`, `uint16`, or `double`.

See Also `isbw`, `isgray`, `isind`

Purpose Convert a label matrix into an RGB image

Syntax

```
RGB = label2rgb(L)
RGB = label2rgb(L,map)
RGB = label2rgb(L,map,zerocolor)
RGB = label2rgb(L,map,zerocolor,order)
```

Description

`RGB = label2rgb(L)` converts a label matrix `L`, such as those returned by `bwlabel` or `watershed`, into an RGB color image for the purpose of visualizing the labeled regions. The `label2rgb` function determines the color to assign to each object based on the number of objects in the label matrix and range of colors in the colormap. The `label2rgb` function picks colors from the entire range.

`RGB = label2rgb(L,map)` defines the colormap to be used in the RGB image. The colormap, `map`, can have any of the following values:

- n-by-3 colormap matrix
- String containing the name of a MATLAB colormap function, such as `'jet'` or `'gray'` (See `colormap` for a list of supported colormaps.)
- Function handle of a colormap function, such as `@jet` or `@gray`

If you do not specify `map`, the default value is `'colorcube'`.

`RGB = label2rgb(L,map,zerocolor)` defines the RGB color of the elements labeled zero (i.e., the background) in the input label matrix `L`. As the value of `zerocolor`, specify an RGB triple or one of the strings listed in this table.

Value	Color
`'b'`	blue
`'c'`	cyan
`'g'`	green
`'k'`	black
`'m'`	magenta
`'r'`	red

Value	Color
'w'	white
'y'	yellow

If you do not specify zerocolor, the default value for zero-labeled elements is [1 1 1] (white).

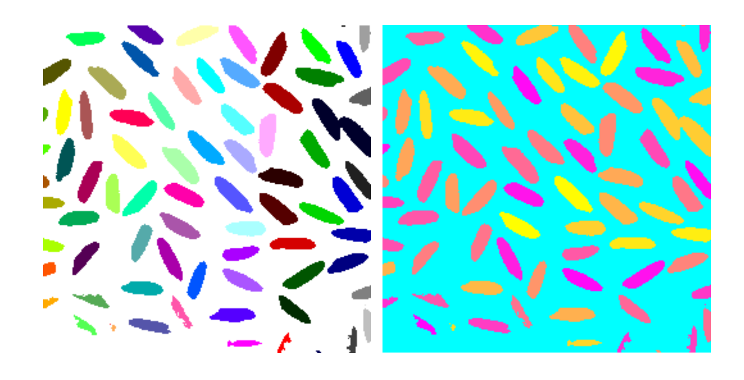
RGB = label2rgb(L,map,zerocolor,order) controls how label2rgb assigns colormap colors to regions in the label matrix. If order is 'noshuffle' (the default), label2rgb assigns colormap colors to label matrix regions in numerical order. If order is 'shuffle', label2rgb assigns colormap colors pseudo-randomly.

Class Support

The input label matrix L can have any nonsparse, numeric class. It must contain finite, nonnegative integers. The output of label2rgb is of class uint8.

Example

```
I = imread('rice.tif');
figure, imshow(I), title('original image')
BW = im2bw(I, graythresh(I));
L = bwlabel(BW);
RGB = label2rgb(L);
RGB2 = label2rgb(L, 'spring', 'c', 'shuffle');
imshow(RGB), figure, imshow(RGB2)
```



See Also

bwlabel, bwlabeln, ismember, watershed

Purpose Construct a lookup table for use with `applylut`

Syntax

```
lut = makelut(fun,n)
lut = makelut(fun,n,P1,P2,...)
```

Description `lut = makelut(fun,n)` returns a lookup table for use with `applylut`. `fun` is either a string containing the name of a function or an inline function object. The function should take a 2-by-2 or 3-by-3 matrix of 1's and 0's as input and return a scalar. `n` is either 2 or 3, indicating the size of the input to `fun`. `makelut` creates `lut` by passing all possible 2-by-2 or 3-by-3 neighborhoods to `fun`, one at a time, and constructing either a 16-element vector (for 2-by-2 neighborhoods) or a 512-element vector (for 3-by-3 neighborhoods). The vector consists of the output from `fun` for each possible neighborhood.

`lut = makelut(fun,n,P1,P2,...)` passes the additional parameters `P1,P2,...,` to `fun`.

Class Support `lut` is returned as a vector of class `double`.

Example In this example, the function returns 1 (true) if the number of 1's in the neighborhood is 2 or greater, and returns 0 (false) otherwise. `makelut` then uses the function to construct a lookup table for 2-by-2 neighborhoods.

```
f = inline('sum(x(:)) >= 2');
lut = makelut(f,2)

lut =

     0
     0
     0
     1
     0
     1
     1
     1
     0
     1
     1
     1
```

```
1
1
1
1
```

See Also `applylut`

Purpose Create resampling structure

Syntax

```
R = makeresampler(interpolant,padmethod)
```

Description

`R = makeresampler(interpolant,padmethod)` creates a separable resampler structure for use with `tformarray` and `imtransform`.

The *`interpolant`* argument specifies the interpolating kernel that the separable resampler uses. In its simplest form, *`interpolant`* can have any of the following strings as a value.

Interpolant	Description
`'cubic'`	Cubic interpolation
`'linear'`	Linear interpolation
`'nearest'`	Nearest neighbor interpolation

If you are using a custom interpolating kernel, you can specify *`interpolant`* as a cell array in either of these forms

`{half_width, positive_half}`	`half_width` is a positive scalar designating the half width of a symmetric interpolating kernel. `positive_half` is a vector of values regularly sampling the kernel on the closed interval `[0 positive_half]`
`{half_width, interp_fcn}`	`interp_fcn` is a function handle that returns interpolating kernel values, given an array of input values in the interval `[0 positive_half]`

To specify the interpolation method independently along each dimension, you can combine both types of interpolant specifications. The number of elements in the cell array must equal the number of transform dimensions. For example, if you specify this value for `interpolant`

```
{'nearest', 'linear', {2 KERNEL_TABLE}}
```

the resampler uses nearest-neighbor interpolation along the first transform dimension, linear interpolation along the second dimension, and a custom table-based interpolation along the third.

The *padmethod* argument controls how the resampler interpolates or assigns values to output elements that map close to or outside the edge of input array. The following table lists all the possible values of *padmethod*.

Pad Method	Description
'bound'	Assigns values from the fill value array to points that map outside the array and repeats border elements of array for points that map inside the array. (Same as 'replicate'.) When interpolant is 'nearest', this pad method produces the same results as 'fill'. 'bound' is like 'fill', but avoids mixing fill values and input image values.
'circular'	Pads array with circular repetition of elements within the dimension. Same as padarray.
'fill'	Generates an output array with smooth-looking edges (except when using nearest neighbor interpolation). For output points that map near the edge of the input array (either inside or outside), it combines input image and fill values. When interpolant is 'nearest', this pad method produces the same results as 'bound'.
'replicate'	Pads array by repeating border elements of array. Same as padarray.
'symmetric'	Pads array with mirror reflections of itself. Same as padarray.

In the case of 'fill', 'replicate', 'circular', or 'symmetric', the resampling performed by tformarray or imtransform occurs in two logical steps:

1. Pad the array A infinitely to fill the entire input transform space

2 Evaluate the convolution of the padded `A` with the resampling kernel at the output points specified by the geometric map

Each nontransform dimension is handled separately. The padding is virtual, (accomplished by remapping array subscripts) for performance and memory efficiency. If you implement a custom resampler, you can implement these behaviors.

Custom Resamplers

The syntaxes described above construct a resampler structure that uses the separable resampler function that ships with the Image Processing Toolbox. It is also possible to create a resampler structure that uses a user-written resampler by using this syntax

```
R = makeresampler(PropertyName,PropertyValue,...)
```

The `makeresampler` function supports the following properties.

Property	Description
`'Type'`	Can have the values `'separable'` or `'custom'` and must always be supplied. If `'Type'` is `'separable'`, the only other properties that can be specified are `'Interpolant'` and `'PadMethod'`, and the result is equivalent to using the `makeresampler(`*`interpolant`*`,`*`padmethod`*`)` syntax. If `'Type'` is `'custom'`, you must specify the `'NDims'` and `'ResampleFcn'` properties and, optionally, the `'CustomData'` property.
`'PadMethod'`	See the padmethod argument for more information.
`'Interpolant'`	See the interpolant argument for more information.
`'NDims'`	`'NDims'` is a positive integer and indicates what dimensionality the custom resampler can handle. Use a value of `Inf` to indicate that the custom resampler can handle any dimension. If `'Type'` is `'custom'`, `NDims` is required.

Property	Description
'ResampleFcn'	This property is a handle to a function that performs the resampling. The function is called with the following interface. `B = resample_fcn(A,M,TDIMS_A,TDIMS_B,FSIZE_A,FSIZE_B,F,R)` See the help for `tformarray` for information about the inputs `A`, `TDIMS_A`, `TDIMS_B`, and `F`. The argument `M` is an array that maps the transform subscript space of `B` to the transform subscript space of `A`. If `A` has `N` transform dimensions (`N=length(TDIMS_A)`) and `B` has `P` transform dimensions (`P=length(TDIMS_B)`), then `ndims(M)=P+1`, if `N>1` and `P` if `N==1`, and `size(M,P+1)=N`. The first `P` dimensions of `M` correspond to the output transform space, permuted according to the order in which the output transform dimensions are listed in `TDIMS_B`. (In general `TDIMS_A` and `TDIMS_B` need not be sorted in ascending order, although such a limitation may be imposed by specific resamplers.) Thus, the first `P` elements of `size(M)` determine the sizes of the transform dimensions of `B`. The input transform coordinates to which each point is mapped are arrayed across the final dimension of `M`, following the order given in `TDIMS_A`. `M` must be double. `FSIZE_A` and `FSIZE_B` are the full sizes of `A` and `B`, padded with `1`'s as necessary to be consistent with `TDIMS_A`, `TDIMS_B`, and `size(A)`.
'CustomData'	User-defined.

Example

Stretch an image in the *y*-direction using separable resampler that applies in cubic interpolation in the *y*-direction and nearest neighbor interpolation in the *x*-direction. (This is equivalent to, but faster than, applying bicubic interpolation.)

```
A = imread('moon.tif');
resamp = makeresampler({'nearest','cubic'},'fill');
stretch = maketform('affine',[1 0; 0 1.3; 0 0]);
B = imtransform(A,stretch,resamp);
```

See Also

`imtransform`, `tformarray`

Purpose

Create geometric transformation structure

Syntax

`T = maketform(`*`transformtype`*`,...)`

Description

`T = maketform(`*`transformtype`*`,...)` creates a multidimensional spatial transformation structure (called a `TFORM` struct) that can be used with the `tformfwd`, `tforminv`, `fliptform`, `imtransform`, or `tformarray` functions.

`transformtype` can be any of the following spatial transformation types. `maketform` supports a special syntax for each transformation type. See the following sections for information about these syntaxes.

Transform Type	Description
`'affine'`	Affine transformation in 2-D or N-D
`'projective'`	Projective transformation in 2-D or N-D
`'custom'`	User-defined transformation, may be N-D to M-D
`'box'`	Independent affine transformation (scale and shift) in each dimension
`'composite'`	Composition of an arbitrary number of more basic transformations

Transform Types

Affine

`T = maketform('affine',A)` builds a `TFORM` struct, `T`, for an N-dimensional affine transformation. `A` is a nonsingular real (N+1)-by-(N+1) or (N+1)-by-N matrix. If `A` is (N+1)-by-(N+1), the last column of `A` must be `[zeros(N,1);1]`. Otherwise, `A` is augmented automatically, such that its last column is `[zeros(N,1);1]`. The matrix `A` defines a forward transformation such that `tformfwd(U,T)`, where `U` is a 1-by-N vector, returns a 1-by-N vector `X`, such that `X=U*A(1:N,1:N)+A(N+1,1:N)`. `T` has both forward and inverse transformations.

`T = maketform('affine',U,X)` builds a `TFORM` struct, `T`, for a two-dimensional affine transformation that maps each row of `U` to the corresponding row of `X`. The `U` and `X` arguments are each 3-by-2 and define the corners of input and output triangles. The corners may not be collinear.

Projective

T = maketform('projective',A) builds a TFORM struct for an N-dimensional projective transformation. A is a nonsingular real (N+1)-by-(N+1) matrix. A(N+1,N+1) cannot be 0. The matrix A defines a forward transformation such that tformfwd(U,T), where U is a 1-by-N vector, returns a 1-by-N vector X, such that X = W(1:N)/W(N+1), where W=[U 1]*A. The transformation structure, T, has both forward and inverse transformations.

T = maketform('projective',U,X) builds a TFORM struct, T, for a two-dimensional projective transformation that maps each row of U to the corresponding row of X. The U and X arguments are each 4-by-2 and define the corners of input and output quadrilaterals. No three corners may be collinear.

Custom

T = maketform('custom',NDIMS_IN,NDIMS_OUT,...
FORWARD_FCN,INVERSE_FCN,TDATA) builds a custom TFORM struct, T, based on user-provided function handles and parameters. NDIMS_IN and NDIMS_OUT are the numbers of input and output dimensions. FORWARD_FCN and INVERSE_FCN are function handles to forward and inverse functions. Those functions must support the syntaxes:

Forward function:	X = FORWARD_FCN(U,T)
Inverse function	U = INVERSE_FCN(X,T)

where U is a P-by-NDIMS_IN matrix whose rows are points in the transformation's input space, and X is a P-by-NDIMS_OUT matrix whose rows are points in the transformation's output space. The TDATA argument can be any MATLAB array and is typically used to store parameters of the custom transformation. It is accessible to FORWARD_FCN and INVERSE_FCN via the “tdata” field of T. Either FORWARD_FCN or INVERSE_FCN can be empty, although at least INVERSE_FCN must be defined to use T with tformarray or imtransform.

Box

T = maketform('box',tsize,LOW,HIGH) or
T = maketform('box',INBOUNDS, OUTBOUNDS) builds an N-dimensional affine TFORM struct, T. The tsize argument is an N-element vector of positive integers. LOW and HIGH are also N-element vectors. The transformation maps

an input box defined by the opposite corners `ones(1,N)` and `tsize` or, alternatively, by corners `INBOUNDS(1,:)` and `INBOUND(2,:)` to an output box defined by the opposite corners `LOW` and `HIGH` or `OUTBOUNDS(1,:)` and `OUTBOUNDS(2,:)`. `LOW(K)` and `HIGH(K)` must be different unless `tsize(K)` is 1, in which case the affine scale factor along the K-th dimension is assumed to be 1.0. Similarly, `INBOUNDS(1,K)` and `INBOUNDS(2,K)` must be different unless `OUTBOUNDS(1,K)` and `OUTBOUNDS(1,K)` are the same, and vice versa. The `'box'` `TFORM` is typically used to register the row and column subscripts of an image or array to some "world" coordinate system.

Composite

`T = maketform('composite',T1,T2,...,TL)` or `T = maketform('composite', [T1 T2 ... TL])` builds a `TFORM` struct, `T`, whose forward and inverse functions are the functional compositions of the forward and inverse functions of the `T1, T2, ..., TL`.

For example, if `L = 3`, then `tformfwd(U,T)` is the same as `tformfwd(tformfwd(tformfwd(U,T3),T2),T1)`. The components `T1` through `TL` must be compatible in terms of the numbers of input and output dimensions. `T` has a defined forward transform function only if all of the component transforms have defined forward transform functions. `T` has a defined inverse transform function only if all of the component functions have defined inverse transform functions.

Example

Make and apply an affine transformation.

```
T = maketform('affine',[.5 0 0; .5 2 0; 0 0 1]);
tformfwd([10 20],T)
I = imread('cameraman.tif');
I2 = imtransform(I,T);
imshow(I2)
```

See Also

`tformfwd`, `tforminv`, `fliptform`, `imtransform`, `tformarray`

mat2gray

Purpose Convert a matrix to a grayscale intensity image

Syntax

```
I = mat2gray(A,[amin amax])
I = mat2gray(A)
```

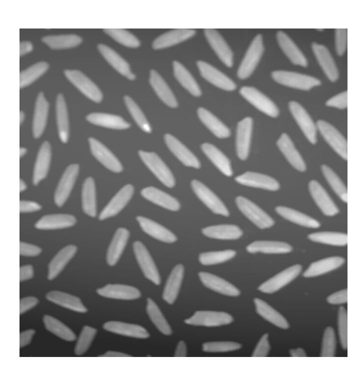
Description `I = mat2gray(A,[amin amax])` converts the matrix `A` to the intensity image `I`. The returned matrix `I` contains values in the range 0 (black) to 1.0 (full intensity or white). `amin` and `amax` are the values in `A` that correspond to 0 and 1.0 in `I`.

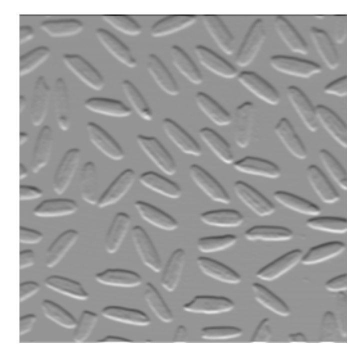
`I = mat2gray(A)` sets the values of `amin` and `amax` to the minimum and maximum values in `A`.

Class Support The input array, `A`, and the output image, `I`, are of class `double`.

Example

```
I = imread('rice.tif');
J = filter2(fspecial('sobel'),I);
K = mat2gray(J);
imshow(I)
figure, imshow(K)
```





See Also `gray2ind`

mean2

Purpose Compute the mean of the elements of a matrix

Syntax

```
B = mean2(A)
```

Description `B = mean2(A)` computes the mean of the values in `A`.

Class Support The input image, `A`, can be numeric or logical. The output image, `B`, is a scalar of class `double`.

Algorithm `mean2` computes the mean of an array `A` using `mean(A(:))`.

See Also `std2`

`mean`, `std` in the MATLAB Function Reference

medfilt2

Purpose Perform two-dimensional median filtering

Syntax

```
B = medfilt2(A,[m n])
B = medfilt2(A)
B = medfilt2(A,'indexed',...)
```

Description Median filtering is a nonlinear operation often used in image processing to reduce “salt and pepper” noise. Median filtering is more effective than convolution when the goal is to simultaneously reduce noise and preserve edges.

`B = medfilt2(A,[m n])` performs median filtering of the matrix `A` in two dimensions. Each output pixel contains the median value in the `m`-by-`n` neighborhood around the corresponding pixel in the input image. `medfilt2` pads the image with zeros on the edges, so the median values for the points within `[m n]/2` of the edges may appear distorted.

`B = medfilt2(A)` performs median filtering of the matrix `A` using the default 3-by-3 neighborhood.

`B = medfilt2(A,'indexed',...)` processes `A` as an indexed image, padding with zeros if the class of `A` is `uint8`, or ones if the class of `A` is `double`.

Class Support The input image, `A`, can be of class `logical`, `uint8`, `uint16`, or `double` (unless the `'indexed'` syntax is used, in which case `A` cannot be of class `uint16`). The output image, `B`, is of the same class as `A`.

Remarks If the input image `A` is of an integer class, all of the output values are returned as integers. If the number of pixels in the neighborhood (i.e., `m*n`) is even, some of the median values may not be integers. In these cases, the fractional parts are discarded. Logical input is treated similarly.

For example, suppose you call `medfilt2` using 2-by-2 neighborhoods, and the input image is a `uint8` array that includes this neighborhood.

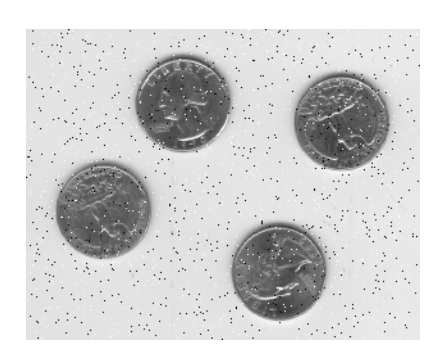
```
1 5
4 8
```

`medfilt2` returns an output value of 4 for this neighborhood, although the true median is 4.5.

Example

This example adds salt and pepper noise to an image, then restores the image using `medfilt2`.

```
I = imread('eight.tif');
J = imnoise(I,'salt & pepper',0.02);
K = medfilt2(J);
imshow(J)
figure, imshow(K)
```





Algorithm

`medfilt2` uses `ordfilt2` to perform the filtering.

See Also

`filter2`, `ordfilt2`, `wiener2`

Reference

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 469-476.

montage

Purpose Display multiple image frames as a rectangular montage

Syntax

```
montage(I)
montage(BW)
montage(X,map)
montage(RGB)
h = montage(...)
```

Description

`montage` displays all of the frames of a multiframe image array in a single image object, arranging the frames so that they roughly form a square.

`montage(I)` displays the `k` frames of the intensity image array `I`. `I` is `m`-by-`n`-by-1-by-`k`.

`montage(BW)` displays the `k` frames of the binary image array `BW`. `BW` is `m`-by-`n`-by-1-by-`k`.

`montage(X,map)` displays the `k` frames of the indexed image array `X`, using the colormap `map` for all frames. `X` is `m`-by-`n`-by-1-by-`k`.

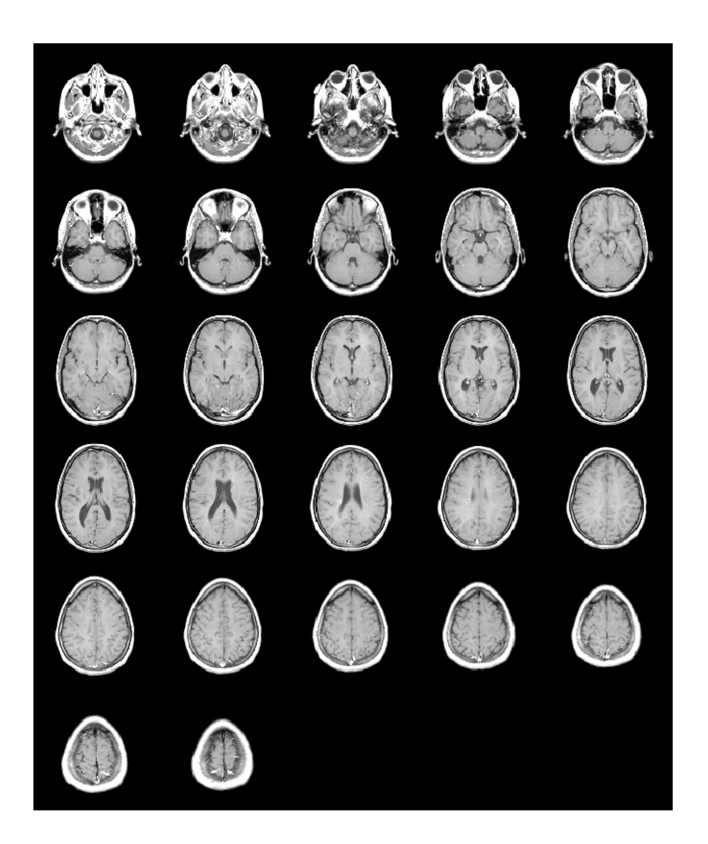
`montage(RGB)` displays the `k` frames of the truecolor image array `RGB`. `RGB` is `m`-by-`n`-by-3-by-`k`.

`h = montage(...)` returns the handle to the image object.

Class Support The input image can be of class `logical`, `uint8`, `uint16`, or `double`.

Example

```
load mri
montage(D,map)
```



See Also immovie

nlfilter

Purpose Perform general sliding-neighborhood operations

Syntax

```
B = nlfilter(A,[m n],fun)
B = nlfilter(A,[m n],fun,P1,P2,...)
B = nlfilter(A,'indexed',...)
```

Description `B = nlfilter(A,[m n],fun)` applies the function `fun` to each `m`-by-`n` sliding block of `A`. `fun` is a function that accepts an `m`-by-`n` matrix as input, and returns a scalar result.

```
c = fun(x)
```

`c` is the output value for the center pixel in the `m`-by-`n` block `x`. `nlfilter` calls `fun` for each pixel in `A`. `nlfilter` zero pads the `m`-by-`n` block at the edges, if necessary.

`B = nlfilter(A,[m n],fun,P1,P2,...)` passes the additional parameters `P1,P2,...,` to `fun`.

`B = nlfilter(A,'indexed',...)` processes `A` as an indexed image, padding with ones if `A` is of class `double` and zeros if `A` is of class `uint8`.

Class Support The input image, `A`, can be of any class supported by `fun`. The class of `B` depends on the class of the output from `fun`.

Remarks `nlfilter` can take a long time to process large images. In some cases, the `colfilt` function can perform the same operation much faster.

Example `fun` can be a `function_handle`, created using `@`. This example produces the same result as calling `medfilt2` with a 3-by-3 neighborhood.

```
B = nlfilter(A,[3 3],@myfun);
```

where `myfun` is an M-file containing

```
function scalar = myfun(x)
scalar = median(x(:));
```

fun can also be an inline object. The example above can be written as

```
fun = inline('median(x(:))');
```

See Also `blkproc`, `colfilt`

normxcorr2

Purpose Normalized two-dimensional cross-correlation

Syntax

```
C = normxcorr2(TEMPLATE,A)
```

Description `C = normxcorr2(TEMPLATE,A)` computes the normalized cross-correlation of the matrices `TEMPLATE` and `A`. The matrix `A` must be larger than the matrix `TEMPLATE` for the normalization to be meaningful. The values of `TEMPLATE` cannot all be the same. The resulting matrix, `C`, contains the correlation coefficients, which may range in value from -1.0 to 1.0.

Class Support The input matrices can be of class `uint8`, `uint16`, or `double`.

Algorithm `normxcorr2` uses the following general procedure:

1. Calculate cross-correlation in the spatial or the frequency domain, depending on size of images.
2. Calculate local sums by precomputing running sums. [1]
3. Use local sums to normalize the cross-correlation to get correlation coefficients. [2]

Example

```
T = .2*ones(11); % make light gray plus on dark gray background
T(6,3:9) = .6;
T(3:9,6) = .6;
BW = T>0.5;      % make white plus on black background
imshow(BW), title('Binary')
figure, imshow(T), title('Template')

% make new image that offsets template T
T_offset = .2*ones(21);
offset = [3 5];  % shift by 3 rows, 5 columns
T_offset( (1:size(T,1))+offset(1), (1:size(T,2))+offset(2) ) = T;
figure, imshow(T_offset), title('Offset Template')

% cross-correlate BW and T_offset to recover offset
cc = normxcorr2(BW,T_offset);
[max_cc, imax] = max(abs(cc(:)));
[ypeak, xpeak] = ind2sub(size(cc),imax(1));
corr_offset = [ (ypeak-size(T,1)) (xpeak-size(T,2)) ];
isequal(corr_offset,offset) % 1 means offset was recovered
```

See Also `corrcoef`

References

[1] J. P. Lewis, "Fast Normalized Cross-Correlation", Industrial Light & Magic.
`<http://www.idiom.com/~zilla/Papers/nvisionInterface/nip.html>`

[2] Robert M. Haralick and Linda G. Shapiro, *Computer and Robot Vision*, Volume II, Addison-Wesley, 1992, pp. 316-317.

Purpose

Convert NTSC values to RGB color space

Syntax

```
rgbmap = ntsc2rgb(yiqmap)
RGB = ntsc2rgb(YIQ)
```

Description

`rgbmap = ntsc2rgb(yiqmap)` converts the `m`-by-3 NTSC (television) color values in `yiqmap` to RGB color space. If `yiqmap` is `m`-by-3 and contains the NTSC luminance *(Y)* and chrominance *(I* and *Q)* color components as columns, then `rgbmap` is an `m`-by-3 matrix that contains the red, green, and blue values equivalent to those colors. Both `rgbmap` and `yiqmap` contain intensities in the range 0 to 1.0. The intensity 0 corresponds to the absence of the component, while the intensity 1.0 corresponds to full saturation of the component.

`RGB = ntsc2rgb(YIQ)` converts the NTSC image `YIQ` to the equivalent truecolor image `RGB`.

`ntsc2rgb` computes the RGB values from the NTSC components using

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Class Support

The input image or colormap must be of class `double`. The output is of class `double`.

See Also

`rgb2ntsc`, `rgb2ind`, `ind2rgb`, `ind2gray`

ordfilt2

Purpose

Perform two-dimensional order-statistic filtering

Syntax

```
B = ordfilt2(A,order,domain)
B = ordfilt2(A,order,domain,S)
B = ordfilt2(...,padopt)
```

Description

`B = ordfilt2(A,order,domain)` replaces each element in `A` by the `order`-th element in the sorted set of neighbors specified by the nonzero elements in `domain`.

`B = ordfilt2(A,order,domain,S)`, where `S` is the same size as `domain`, uses the values of `S` corresponding to the nonzero values of `domain` as additive offsets.

`B = ordfilt2(...,padopt)` controls how the matrix boundaries are padded. Set `padopt` to `'zeros'` (the default), or `'symmetric'`. If `padopt` is `'zeros'`, `A` is padded with zeros at the boundaries. If `padopt` is `'symmetric'`, `A` is symmetrically extended at the boundaries.

Class Support

The class of `A` can be `logical`, `uint8`, `uint16`, or `double`. The class of `B` is the same as the class of `A`, unless the additive offset form of `ordfilt2` is used, in which case the class of `B` is `double`.

Remarks

`domain` is equivalent to the structuring element used for binary image operations. It is a matrix containing only 1's and 0's; the 1's define the neighborhood for the filtering operation.

For example, `B = ordfilt2(A,5,ones(3,3))` implements a 3-by-3 median filter; `B = ordfilt2(A,1,ones(3,3))` implements a 3-by-3 minimum filter; and `B = ordfilt2(A,9,ones(3,3))` implements a 3-by-3 maximum filter. `B = ordfilt2(A,1,[0 1 0; 1 0 1; 0 1 0])` replaces each element in `A` by the minimum of its north, east, south, and west neighbors.

The syntax that includes `S` (the matrix of additive offsets) can be used to implement grayscale morphological operations, including grayscale dilation and erosion.

See Also

`medfilt2`

Reference

[1] Haralick, Robert M., and Linda G. Shapiro. *Computer and Robot Vision, Volume I.* Addison-Wesley, 1992.

otf2psf

Purpose Convert optical transfer function to point-spread function

Syntax

```
PSF = otf2psf(OTF)
PSF = otf2psf(OTF,OUTSIZE)
```

Description

PSF = otf2psf(OTF) computes the inverse Fast Fourier Transform (IFFT) of the optical transfer function (OTF) array and creates a point-spread function (PSF), centered at the origin. By default, the PSF is the same size as the OTF.

PSF = otf2psf(OTF,OUTSIZE) converts the OTF array into a PSF array, where OUTSIZE specifies the size of the output point-spread function. The size of the output array must not exceed the size of the OTF array in any dimension.

To center the PSF at the origin, otf2psf circularly shifts the values of the output array down (or to the right) until the (1,1) element reaches the central position, then it crops the result to match dimensions specified by OUTSIZE.

Note that this function is used in image convolution/deconvolution when the operations involve the FFT.

Class Support OTF can be any nonsparse, numeric array. PSF is of class double.

Example

```
PSF  = fspecial('gaussian',13,1);
OTF  = psf2otf(PSF,[31 31]); % PSF --> OTF
PSF2 = otf2psf(OTF,size(PSF)); % OTF --> PSF2
subplot(1,2,1); surf(abs(OTF)); title('|OTF|');
axis square; axis tight
subplot(1,2,2); surf(PSF2); title('corresponding PSF');
axis square; axis tight
```

See Also psf2otf, circshift, padarray

Purpose Pad an array

Syntax

```
B = padarray(A,padsize,method,direction)
```

Description

`B = padarray(A,padsize,method,direction)` pads array `A` with `padsize` number of elements along each dimension, using specified `method` and `direction`. The `N`-th element of the `padsize` vector specifies the padding size for the `N`-th dimension of the array, `A`.

`method` can be a numeric scalar, in which case it specifies the value for all padding elements, or one of the following strings which specify other methods used to determine pad values. By default, `METHOD` sets the pad value to the `0`.

Value	Meaning
'circular'	Pad with circular repetition of elements within the dimension.
'replicate'	Pad by repeating border elements of array.
'symmetric'	Pad array with mirror reflections of itself.

`direction` specifies where to pad the array along each dimension. By default, `direction` is `'post'`.

Value	Meaning
'pre'	Pad before the first element of each dimension.
'post'	Pad after the last element of each dimension. This is the default.
'both'	Pads before the first element and after the last element of each dimension.

Class Support

When padding with a constant value, `A` can be numeric or logical. When padding using the `'circular'`, `'replicate'`, or `'symmetric'` methods, `A` can be of any class. `B` is of the same class as `A`.

Example

Add three elements of padding to the beginning of a vector. The padding elements contain mirror copies of the array.

```
b = padarray([1 2 3 4],3,'symmetric','pre')
b =

     3     2     1     1     2     3     4
```

Add three elements of padding to the end of the first dimension of the array and two elements of padding to the end of the second dimension. Use the value of the last array element as the padding value.

```
B = padarray([1 2; 3 4],[3 2],'replicate','post')
B =

     1     2     2     2
     3     4     4     4
     3     4     4     4
     3     4     4     4
     3     4     4     4
```

Add three elements of padding to each dimension of a three-dimensional array. Each pad element contains the value 0.

```
A = [ 1 2; 3 4];
B = [ 5 6; 7 8];
C = cat(3,A,B)
C(:,:,1) =

     1     2
     3     4

C(:,:,2) =

     5     6
     7     8

D = padarray(C,[3 3],0,'both')
```

```
D(:,:,1) =
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     1     2     0     0     0
     0     0     0     3     4     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0

D(:,:,2) =
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     5     6     0     0     0
     0     0     0     7     8     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

See Also

`circshift`, `imfilter`

phantom

Purpose

Generate a head phantom image

Syntax

```
P = phantom(def,n)
P = phantom(E,n)
[P,E] = phantom(...)
```

Description

`P = phantom(def,n)` generates an image of a head phantom that can be used to test the numerical accuracy of `radon` and `iradon` or other two-dimensional reconstruction algorithms. `P` is a grayscale intensity image that consists of one large ellipse (representing the brain) containing several smaller ellipses (representing features in the brain).

`def` is a string that specifies the type of head phantom to generate. Valid values are:

- `'Shepp-Logan'` — a test image used widely by researchers in tomography.
- `'Modified Shepp-Logan'` (default) — a variant of the Shepp-Logan phantom in which the contrast is improved for better visual perception.

`n` is a scalar that specifies the number of rows and columns in `P`. If you omit the argument, `n` defaults to 256.

`P = phantom(E,n)` generates a user-defined phantom, where each row of the matrix `E` specifies an ellipse in the image. `E` has six columns, with each column containing a different parameter for the ellipses. This table describes the columns of the matrix.

Column	Parameter	Meaning
Column 1	`A`	Additive intensity value of the ellipse
Column 2	`a`	Length of the horizontal semi-axis of the ellipse
Column 3	`b`	Length of the vertical semi-axis of the ellipse
Column 4	`x0`	*x*-coordinate of the center of the ellipse

Column	Parameter	Meaning
Column 5	`y0`	*y*-coordinate of the center of the ellipse
Column 6	`phi`	Angle (in degrees) between the horizontal semi-axis of the ellipse and the *x*-axis of the image

For purposes of generating the phantom, the domains for the *x*- and *y*-axes span [–1,1]. Columns 2 through 5 must be specified in terms of this range.

`[P,E] = phantom(...)` returns the matrix `E` used to generate the phantom.

Class Support

All inputs and all outputs must be of class `double`.

Remarks

For any given pixel in the output image, the pixel's value is equal to the sum of the additive intensity values of all ellipses that the pixel is a part of. If a pixel is not part of any ellipse, its value is 0.

The additive intensity value `A` for an ellipse can be positive or negative; if it is negative, the ellipse will be darker than the surrounding pixels. Note that, depending on the values of `A`, some pixels may have values outside the range [0,1].

Example

```
P = phantom('Modified Shepp-Logan',200);
imshow(P)
```



Reference

[1] Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989. p. 439.

See Also

`radon`, `iradon`

Purpose Display information about image pixels

Syntax

```
pixval on
pixval off
pixval
pixval(fig,option)
```

Purpose

`pixval on` turns on interactive display of information about image pixels in the current figure. `pixval` installs a black bar at the bottom of the figure, which displays the (x,y) coordinates for whatever pixel the cursor is currently over, and the color information for that pixel. If the image is binary or intensity, the color information is a single intensity value. If the image is indexed or RGB, the color information is an RGB triplet. The values displayed are the actual data values, regardless of the class of the image array, or whether the data is in normal image range.

If you click on the image and hold down the mouse button while you move the cursor, `pixval` also displays the Euclidean distance between the point you clicked on and the current cursor location. `pixval` draws a line between these points to indicate the distance being measured. When you release the mouse button, the line and the distance display disappear.

You can move the display bar by clicking on it and dragging it to another place in the figure.

`pixval off` turns interactive display off in the current figure. You can also turn off the display by clicking the button on the right side of the display bar.

`pixval` toggles interactive display on or off in the current figure.

`pixval(fig,option)` applies the `pixval` command to the figure specified by `fig`. `option` is string containing `'on'` or `'off'`.

See Also `impixel`, `improfile`

psf2otf

Purpose

Convert point-spread function to optical transfer function.

Syntax

```
OTF = psf2otf(PSF)
OTF = psf2otf(PSF,OUTSIZE)
```

Description

`OTF = psf2otf(PSF)` computes the Fast Fourier Transform (FFT) of the point-spread function (`PSF`) array and creates the optical transfer function array, `OTF`, that is not influenced by the `PSF` off-centering. By default, the `OTF` array is the same size as the `PSF` array.

`OTF = psf2otf(PSF,OUTSIZE)` converts the `PSF` array into an `OTF` array, where `OUTSIZE` specifies the size of the `OTF` array. The `OUTSIZE` cannot be smaller than the `PSF` array size in any dimension.

To ensure that the `OTF` is not altered due to `PSF` off-centering, `psf2otf` post-pads the `PSF` array (down or to the right) with zeros to match dimensions specified in `OUTSIZE`, then circularly shifts the values of the `PSF` array up (or to the left) until the central pixel reaches (1,1) position.

Note that this function is used in image convolution/deconvolution when the operations involve the `FFT`.

Class Support

`PSF` can be any nonsparse, numeric array. `OTF` is of class `double`.

Example

```
PSF  = fspecial('gaussian',13,1);
OTF  = psf2otf(PSF,[31 31]); % PSF --> OTF
subplot(1,2,1); surf(PSF); title('PSF');
axis square; axis tight
subplot(1,2,2); surf(abs(OTF)); title('corresponding |OTF|');
axis square; axis tight
```

See Also

`otf2psf`, `circshift`, `padarray`

Purpose

Perform quadtree decomposition

Syntax

```
S = qtdecomp(I)
S = qtdecomp(I,threshold)
S = qtdecomp(I,threshold,mindim)
S = qtdecomp(I,threshold,[mindim maxdim])

S = qtdecomp(I,fun)
S = qtdecomp(I,fun,P1,P2,...)
```

Description

`qtdecomp` divides a square image into four equal-sized square blocks, and then tests each block to see if it meets some criterion of homogeneity. If a block meets the criterion, it is not divided any further. If it does not meet the criterion, it is subdivided again into four blocks, and the test criterion is applied to those blocks. This process is repeated iteratively until each block meets the criterion. The result may have blocks of several different sizes.

`S = qtdecomp(I)` performs a quadtree decomposition on the intensity image `I`, and returns the quadtree structure in the sparse matrix `S`. If `S(k,m)` is nonzero, then `(k,m)` is the upper-left corner of a block in the decomposition, and the size of the block is given by `S(k,m)`. By default, `qtdecomp` splits a block unless all elements in the block are equal.

`S = qtdecomp(I,threshold)` splits a block if the maximum value of the block elements minus the minimum value of the block elements is greater than `threshold`. `threshold` is specified as a value between 0 and 1, even if `I` is of class `uint8` or `uint16`. If `I` is `uint8`, the threshold value you supply is multiplied by 255 to determine the actual threshold to use; if `I` is `uint16`, the threshold value you supply is multiplied by 65535.

`S = qtdecomp(I,threshold,mindim)` will not produce blocks smaller than `mindim`, even if the resulting blocks do not meet the threshold condition.

`S = qtdecomp(I,threshold,[mindim maxdim])` will not produce blocks smaller than `mindim` or larger than `maxdim`. Blocks larger than `maxdim` are split even if they meet the threshold condition. `maxdim/mindim` must be a power of 2.

`S = qtdecomp(I,fun)` uses the function `fun` to determine whether to split a block. `qtdecomp` calls `fun` with all the current blocks of size `m`-by-`m` stacked into an `m`-by-`m`-by-`k` array, where `k` is the number of `m`-by-`m` blocks. `fun` should return

a k-element vector, containing only 1's and 0's, where 1 indicates that the corresponding block should be split, and 0 indicates it should not be split. (For example, if k(3) is 0, the third m-by-m block should not be split.) fun can be a function_handle, created using @, or an inline object.

S = qtdecomp(I,fun,P1,P2,...) passes P1,P2,..., as additional arguments to fun.

Class Support

For the syntaxes that do not include a function, the input image can be of class logical, uint8, uint16, or double. For the syntaxes that include a function, the input image can be of any class supported by the function. The output matrix is always of class sparse.

Remarks

qtdecomp is appropriate primarily for square images whose dimensions are a power of 2, such as 128-by-128 or 512-by-512. These images can be divided until the blocks are as small as 1-by-1. If you use qtdecomp with an image whose dimensions are not a power of 2, at some point the blocks cannot be divided further. For example, if an image is 96-by-96, it can be divided into blocks of size 48-by-48, then 24-by-24, 12-by-12, 6-by-6, and finally 3-by-3. No further division beyond 3-by-3 is possible. To process this image, you must set mindim to 3 (or to 3 times a power of 2); if you are using the syntax that includes a function, the function must return 0 at the point when the block cannot be divided further.

Example

```
I = [1     1     1     1     2     3     6     6
     1     1     2     1     4     5     6     8
     1     1     1     1    10    15     7     7
     1     1     1     1    20    25     7     7
    20    22    20    22     1     2     3     4
    20    22    22    20     5     6     7     8
    20    22    20    20     9    10    11    12
    22    22    20    20    13    14    15    16];

S = qtdecomp(I,5);

full(S)
```

```
ans =

     4     0     0     0     2     0     2     0
     0     0     0     0     0     0     0     0
     0     0     0     0     1     1     2     0
     0     0     0     0     1     1     0     0
     4     0     0     0     2     0     2     0
     0     0     0     0     0     0     0     0
     0     0     0     0     2     0     2     0
     0     0     0     0     0     0     0     0
```

See Also `qtgetblk`, `qtsetblk`

qtgetblk

Purpose Get block values in quadtree decomposition

Syntax

```
[vals,r,c] = qtgetblk(I,S,dim)
[vals,idx] = qtgetblk(I,S,dim)
```

Description `[vals,r,c] = qtgetblk(I,S,dim)` returns in `vals` an array containing the `dim`-by-`dim` blocks in the quadtree decomposition of `I`. `S` is the sparse matrix returned by `qtdecomp`; it contains the quadtree structure. `vals` is a `dim`-by-`dim`-by-`k` array, where `k` is the number of `dim`-by-`dim` blocks in the quadtree decomposition; if there are no blocks of the specified size, all outputs are returned as empty matrices. `r` and `c` are vectors containing the row and column coordinates of the upper-left corners of the blocks.

`[vals,idx] = qtgetblk(I,S,dim)` returns in `idx` a vector containing the linear indices of the upper-left corners of the blocks.

Class Support `I` can be of class `logical`, `uint8`, `uint16`, or `double`. `S` is of class `sparse`.

Remarks The ordering of the blocks in `vals` matches the columnwise order of the blocks in `I`. For example, if `vals` is 4-by-4-by-2, `vals(:,:,1)` contains the values from the first 4-by-4 block in `I`, and `vals(:,:,2)` contains the values from the second 4-by-4 block.

Example This example continues the `qtdecomp` example.

```
[vals,r,c] = qtgetblk(I,S,4)

vals(:,:,1) =

     1     1     1     1
     1     1     2     1
     1     1     1     1
     1     1     1     1
```

```
vals(:,:,2) =

    20    22    20    22
    20    22    22    20
    20    22    20    20
    22    22    20    20


r =

     1
     5


c =

     1
     1
```

See Also qtdecomp, qtsetblk

qtsetblk

Purpose Set block values in quadtree decomposition

Syntax

```
J = qtsetblk(I,S,dim,vals)
```

Description J = qtsetblk(I,S,dim,vals) replaces each dim-by-dim block in the quadtree decomposition of I with the corresponding dim-by-dim block in vals. S is the sparse matrix returned by qtdecomp; it contains the quadtree structure. vals is a dim-by-dim-by-k array, where k is the number of dim-by-dim blocks in the quadtree decomposition.

Class Support I can be of class logical, uint8, uint16, or double. S is of class sparse.

Remarks The ordering of the blocks in vals must match the columnwise order of the blocks in I. For example, if vals is 4-by-4-by-2, vals(:,:,1) contains the values used to replace the first 4-by-4 block in I, and vals(:,:,2) contains the values for the second 4-by-4 block.

Example This example continues the qtgetblock example.

```
newvals = cat(3,zeros(4),ones(4));
J = qtsetblk(I,S,4,newvals)

J =

     0     0     0     0     2     3     6     6
     0     0     0     0     4     5     6     8
     0     0     0     0    10    15     7     7
     0     0     0     0    20    25     7     7
     1     1     1     1     1     2     3     4
     1     1     1     1     5     6     7     8
     1     1     1     1     9    10    11    12
     1     1     1     1    13    14    15    16
```

See Also qtdecomp, qtgetblk

Purpose

Radon transform

Syntax

```
R = radon(I,theta)
[R,xp] = radon(...)
```

Description

The radon function computes the Radon transform, which is the projection of the image intensity along a radial line oriented at a specified angle.

`R = radon(I,theta)` returns the Radon transform of the intensity image `I` for the angle `theta` degrees. If `theta` is a scalar, the result `R` is a column vector containing the Radon transform for `theta` degrees. If `theta` is a vector, then `R` is a matrix in which each column is the Radon transform for one of the angles in `theta`. If you omit `theta`, it defaults to 0:179.

`[R,xp] = radon(...)` returns a vector `xp` containing the radial coordinates corresponding to each row of `R`.

Class Support

`I` can be of class `double`, `logical`, or of any integer class. All other inputs and outputs are of class `double`.

Remarks

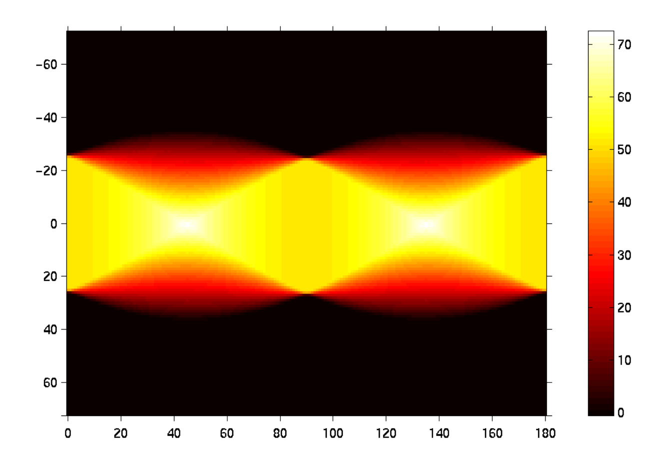
The radial coordinates returned in `xp` are the values along the x'-axis, which is oriented at `theta` degrees counterclockwise from the x-axis. The origin of both axes is the center pixel of the image, which is defined as

```
floor((size(I)+1)/2)
```

For example, in a 20-by-30 image, the center pixel is (10,15).

Example

```
iptsetpref('ImshowAxesVisible','on')
I = zeros(100,100);
I(25:75,25:75) = 1;
theta = 0:180;
[R,xp] = radon(I,theta);
imshow(theta,xp,R,[],'notruesize'), colormap(hot), colorbar
```



See Also `iradon`, `phantom`

References

Bracewell, Ronald N. *Two-Dimensional Imaging*. Englewood Cliffs, NJ: Prentice Hall, 1995. pp. 505-537.

Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 42-45.

Purpose Reflect structuring element

Syntax

```
SE2 = reflect(SE)
```

Description `SE2 = reflect(SE)` reflects a structuring element through its center. The effect is the same as if you rotated the structuring element's domain 180 degrees around its center (for a 2-D structuring element). If `SE` is an array of structuring element objects, then `reflect(SE)` reflects each element of `SE`, and `SE2` has the same size as `SE`.

Class Support `SE` and `SE2` are `STREL` objects.

Example

```
se = strel([0 0 1; 0 0 0; 0 0 0])
se2 = reflect(se)
se =
Flat STREL object containing 1 neighbor.

Neighborhood:
     0     0     1
     0     0     0
     0     0     0

se2 =
Flat STREL object containing 1 neighbor.

Neighborhood:
     0     0     0
     0     0     0
     1     0     0
```

See Also `strel`

regionprops

Purpose Measure properties of image regions

Syntax

```
STATS = regionprops(L,properties)
```

Description

`STATS = regionprops(L,properties)` measures a set of properties for each labeled region in the label matrix `L`. Positive integer elements of `L` correspond to different regions. For example, the set of elements of `L` equal to `1` corresponds to region `1`; the set of elements of `L` equal to `2` corresponds to region `2`; and so on. The return value, `STATS`, is a structure array of length `max(L(:))`. The fields of the structure array denote different measurements for each region, as specified by *`properties`*.

`properties` can be a comma-separated list of strings, a cell array containing strings, the single string `'all'`, or the string `'basic'`. This table lists the set of valid property strings. Property strings are case insensitive and can be abbreviated.

`'Area'`	`'EquivDiameter'`	`'MajorAxisLength'`
`'BoundingBox'`	`'EulerNumber'`	`'MinorAxisLength'`
`'Centroid'`	`'Extent'`	`'Orientation'`
`'ConvexArea'`	`'Extrema'`	`'PixelIdxList'`
`'ConvexHull'`	`'FilledArea'`	`'PixelList'`
`'ConvexImage'`	`'FilledImage'`	`'Solidity'`
`'Eccentricity'`	`'Image'`	`'SubarrayIdx'`

If *`properties`* is the string `'all'`, then all of the above measurements are computed. If *`properties`* is not specified or if it is the string `'basic'`, then these measurements are computed: `'Area'`, `'Centroid'`, and `'BoundingBox'`.

Definitions

`'Area'` – Scalar; the actual number of pixels in the region. (This value may differ slightly from the value returned by `bwarea`, which weights different patterns of pixels differently.)

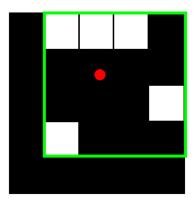
`'Centroid'` – 1-by-`ndims(L)` vector; the center of mass of the region. Note that the first element of `Centroid` is the horizontal coordinate (or *x*-coordinate) of

the center of mass, and the second element is the vertical coordinate (or *y*-coordinate). All other elements of `Centroid` are in order of dimension.

`'BoundingBox'` – 1-by-`ndims(L)*2` vector; the smallest rectangle containing the region. `BoundingBox` is `[ul_corner width]`, where

`ul_corner`	is in the form `[x y z ...]` and specifies the upper-left corner of the bounding box
`width`	is in the form `[x_width y_width ...]` and specifies the width of the bounding box along each dimension

This figure illustrates the centroid and bounding box. The region consists of the white pixels; the green box is the bounding box, and the red dot is the centroid.



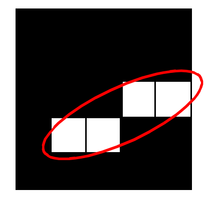
`'MajorAxisLength'` – Scalar; the length (in pixels) of the major axis of the ellipse that has the same second-moments as the region. This property is supported only for 2-D input label matrices.

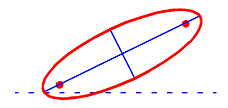
`'MinorAxisLength'` – Scalar; the length (in pixels) of the minor axis of the ellipse that has the same second-moments as the region. This property is supported only for 2-D input label matrices.

`'Eccentricity'` – Scalar; the eccentricity of the ellipse that has the same second-moments as the region. The eccentricity is the ratio of the distance between the foci of the ellipse and its major axis length. The value is between 0 and 1. (0 and 1 are degenerate cases; an ellipse whose eccentricity is 0 is actually a circle, while an ellipse whose eccentricity is 1 is a line segment.) This property is supported only for 2-D input label matrices.

`'Orientation'` – Scalar; the angle (in degrees) between the *x*-axis and the major axis of the ellipse that has the same second-moments as the region. This property is supported only for 2-D input label matrices.

This figure illustrates the axes and orientation of the ellipse. The left side of the figure shows an image region and its corresponding ellipse. The right side shows the same ellipse, with features indicated graphically; the solid blue lines are the axes, the red dots are the foci, and the orientation is the angle between the horizontal dotted line and the major axis.



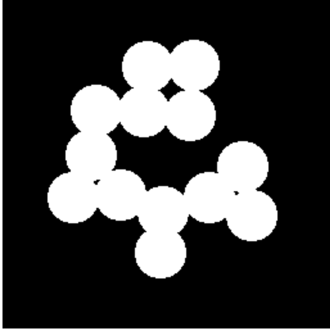


`'Image'` – Binary image (`logical`) of the same size as the bounding box of the region; the `on` pixels correspond to the region, and all other pixels are `off`.

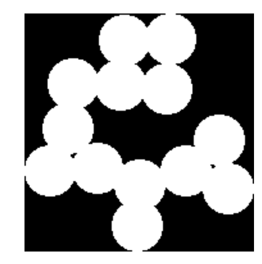
`'FilledImage'` – Binary image (`logical`) of the same size as the bounding box of the region. The `on` pixels correspond to the region, with all holes filled in.

`'FilledArea'` – Scalar; the number of `on` pixels in `FilledImage`.

This figure illustrates `Image'` and `FilledImage'`.



Original image, containing a single region



`'Image'`



`'FilledImage'`

`'ConvexHull'` – p-by-2 matrix; the smallest convex polygon that can contain the region. Each row of the matrix contains the *x*- and *y*-coordinates of one vertex of the polygon. This property is supported only for 2-D input label matrices.

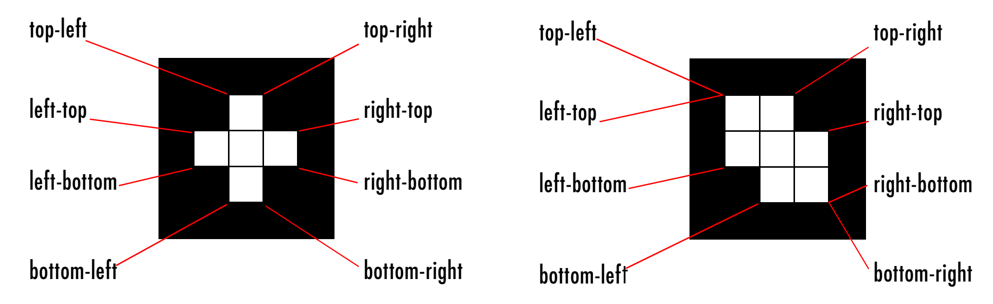
`'ConvexImage'` – Binary image (`logical`); the convex hull, with all pixels within the hull filled in (i.e., set to `on`). (For pixels that the boundary of the hull passes through, `regionprops` uses the same logic as `roipoly` to determine whether the pixel is inside or outside the hull.) The image is the size of the bounding box of the region. This property is supported only for 2-D input label matrices.

`'ConvexArea'` – Scalar; the number of pixels in `'ConvexImage'`. This property is supported only for 2-D input label matrices.

`'EulerNumber'` – Scalar; equal to the number of objects in the region minus the number of holes in those objects. This property is supported only for 2-D input label matrices.

`'Extrema'` – 8-by-2 matrix; the extremal points in the region. Each row of the matrix contains the *x*- and *y*-coordinates of one of the points. The format of the vector is `[top-left top-right right-top right-bottom bottom-right bottom-left left-bottom left-top]`. This property is supported only for 2-D input label matrices.

This figure illustrates the extrema of two different regions. In the region on the left, each extremal point is distinct; in the region on the right, certain extremal points (e.g., `top-left` and `left-top`) are identical.



`'EquivDiameter'` – Scalar; the diameter of a circle with the same area as the region. Computed as `sqrt(4*Area/pi)`. This property is supported only for 2-D input label matrices.

`'Solidity'` – Scalar; the proportion of the pixels in the convex hull that are also in the region. Computed as `Area/ConvexArea`. This property is supported only for 2-D input label matrices.

'Extent' – Scalar; the proportion of the pixels in the bounding box that are also in the region. Computed as the Area divided by area of the bounding box. This property is supported only for 2-D input label matrices.

'PixelList' – p-by-ndims(L) matrix; the actual pixels in the region. Each row of the matrix has the form [x y z ...] and specifies the coordinates of one pixel in the region.

Class Support

The input label matrix L can have any numeric class.

Remarks

The comma-separated list syntax for structure arrays is very useful when working with the output of regionprops. For example, for a field that contains a scalar, you can use a this syntax to create a vector containing the value of this field for each region in the image.

For instance, if stats is a structure array with field Area, then the following two expressions are equivalent

```
stats(1).Area, stats(2).Area, ..., stats(end).Area
```

and

```
stats.Area
```

Therefore, you can use these calls to create a vector containing the area of each region in the image.

```
stats = regionprops(L,'Area');
allArea = [stats.Area];
```

allArea is a vector of the same length as the structure array stats.

The function ismember is useful in conjunction with regionprops for selecting regions based on certain criteria. For example, these commands create a binary image containing only the regions in text.tif whose area is greater than 80.

```
idx = find([stats.Area] > 80);
BW2 = ismember(L,idx);
```

Most of the measurements take very little time to compute. The exceptions are these, which may take significantly longer, depending on the number of regions in L:

- 'ConvexHull'

- 'ConvexImage'
- 'ConvexArea'
- 'FilledImage'

Note that computing certain groups of measurements takes about the same amount of time as computing just one of them because regionprops takes advantage of intermediate computations used in both computations. Therefore, it is fastest to compute all of the desired measurements in a single call to regionprops.

Example

```
BW = imread('text.tif');
L = bwlabel(BW);
stats = regionprops(L,'all');
stats(23)

ans =

                Area: 89
            Centroid: [95.6742 192.9775]
         BoundingBox: [87.5000 184.5000 16 15]
     MajorAxisLength: 19.9127
     MinorAxisLength: 14.2953
        Eccentricity: 0.6961
         Orientation: 9.0845
          ConvexHull: [28x2  double]
         ConvexImage: [15x16 uint8 ]
          ConvexArea: 205
               Image: [15x16 uint8 ]
         FilledImage: [15x16 uint8 ]
          FilledArea: 122
         EulerNumber: 0
             Extrema: [ 8x2  double]
       EquivDiameter: 10.6451
            Solidity: 0.4341
              Extent: 0.3708
           PixelList: [89x2  double]
```

See Also

bwlabel, bwlabeln, ismember, watershed

ismember is a MATLAB function

rgb2gray

Purpose Convert an RGB image or colormap to grayscale

Syntax

```
I = rgb2gray(RGB)
newmap = rgb2gray(map)
```

Description `rgb2gray` converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance.

`I = rgb2gray(RGB)` converts the truecolor image `RGB` to the grayscale intensity image `I`.

`newmap = rgb2gray(map)` returns a grayscale colormap equivalent to `map`.

Class Support If the input is an RGB image, it can be of class `uint8`, `uint16`, or `double`. The output image, `I`, is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class `double`.

Algorithm `rgb2gray` converts the RGB values to NTSC coordinates, sets the hue and saturation components to zero, and then converts back to RGB color space.

See Also `ind2gray`, `ntsc2rgb`, `rgb2ind`, `rgb2ntsc`

Purpose

Convert RGB values to hue-saturation-value (HSV) color space

`rgb2hsv` is a function in MATLAB. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference pages.

rgb2ind

Purpose Convert an RGB image to an indexed image

Syntax

```
[X,map] = rgb2ind(RGB,tol)
[X,map] = rgb2ind(RGB,n)
X = rgb2ind(RGB,map)
[...] = rgb2ind(...,dither_option)
```

Description

`rgb2ind` converts RGB images to indexed images using one of three different methods: uniform quantization, minimum variance quantization, and colormap mapping. For all of these methods, `rgb2ind` also dithers the image unless you specify `'nodither'` for `dither_option`.

`[X,map] = rgb2ind(RGB,tol)` converts the RGB image to an indexed image `X` using uniform quantization. `map` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0 and 1.0.

`[X,map] = rgb2ind(RGB,n)` converts the RGB image to an indexed image `X` using minimum variance quantization. `map` contains at most `n` colors. `n` must be less than or equal to 65536.

`X = rgb2ind(RGB,map)` converts the RGB image to an indexed image `X` with colormap `map` by matching colors in `RGB` with the nearest color in the colormap `map`. `size(map,1)` must be less than or equal to 65536.

`[...] = rgb2ind(...,`*`dither_option`*`)` enables or disables dithering. *`dither_option`* is a string that can have one of these values:

- `'dither'` (default) dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.
- `'nodither'` maps each color in the original image to the closest color in the new map. No dithering is performed.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`. If the length of `map` is less than or equal to 256, the output image is of class `uint8`. Otherwise, the output image is of class `uint16`.

Remarks

If you specify `tol`, `rgb2ind` uses uniform quantization to convert the image. This method involves cutting the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is

```
n = (floor(1/tol)+1)^3
```

Each cube represents a single color in the output image. Therefore, the maximum length of the colormap is n. rgb2ind removes any colors that don't appear in the input image, so the actual colormap may be much smaller than n.

If you specify n, rgb2ind uses minimum variance quantization. This method involves cutting the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number you specify, the output colormap is also smaller.

If you specify map, rgb2ind uses colormap mapping, which involves finding the colors in map that best match the colors in the RGB image.

Example

```
RGB = imread('flowers.tif');
[X,map] = rgb2ind(RGB,128);
imshow(X,map)
```



See Also

cmunique, dither, imapprox, ind2rgb, rgb2gray

rgb2ntsc

Purpose

Convert RGB values to NTSC color space

Syntax

```
yiqmap = rgb2ntsc(rgbmap)
YIQ = rgb2ntsc(RGB)
```

Description

`yiqmap = rgb2ntsc(rgbmap)` converts the `m`-by-3 RGB values in `rbgmap` to NTSC color space. `yiqmap` is an `m`-by-3 matrix that contains the NTSC luminance (Y) and chrominance (I and Q) color components as columns that are equivalent to the colors in the RGB colormap.

`YIQ = rgb2ntsc(RGB)` converts the truecolor image `RGB` to the equivalent NTSC image `YIQ`.

Remarks

In the NTSC color space, the luminance is the grayscale signal used to display pictures on monochrome (black and white) televisions. The other components carry the hue and saturation information.

`rgb2ntsc` defines the NTSC components using

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Class Support

If the input is an RGB image, it can be of class `uint8`, `uint16`, or `double`; the output image is of class `double`. If the input is a colormap, the input and output colormaps are both of class `double`.

See Also

`ntsc2rgb`, `rgb2ind`, `ind2rgb`, `ind2gray`

Purpose Convert RGB values to YCbCr color space

Syntax

```
ycbcrmap = rgb2ycbcr(rgbmap)
YCBCR = rgb2ycbcr(RGB)
```

Description `ycbcrmap = rgb2ycbcr(rgbmap)` converts the RGB values in `rbgmap` to the YCbCr color space. `ycbcrmap` is an `m`-by-3 matrix that contains the YCbCr luminance (*Y*) and chrominance (*Cb* and *Cr*) color components as columns. Each row represents the equivalent color to the corresponding row in the RGB colormap.

`YCBCR = rgb2ycbcr(RGB)` converts the truecolor image `RGB` to the equivalent image in the YCbCr color space.

Class Support If the input is an RGB image, it can be of class `uint8`, `uint16`, or `double`; the output image is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class `double`.

See Also `ntsc2rgb`, `rgb2ntsc`, `ycbcr2rgb`

Purpose Plot colormap

`rgbplot` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose

Select region of interest, based on color

Syntax

```
BW = roicolor(A,low,high)
BW = roicolor(A,v)
```

Description

`roicolor` selects a region of interest within an indexed or intensity image and returns a binary image. (You can use the returned image as a mask for masked filtering using `roifilt2`.)

`BW = roicolor(A,low,high)` returns a region of interest selected as those pixels that lie within the colormap range `[low high]`.

```
BW = (A >= low) & (A <= high)
```

`BW` is a binary image with 0's outside the region of interest and 1's inside.

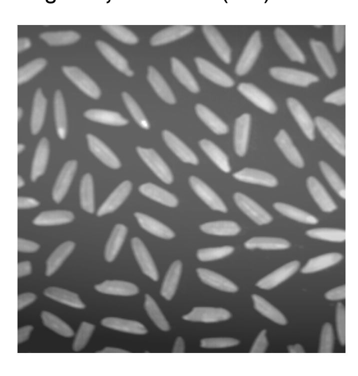
`BW = roicolor(A,v)` returns a region of interest selected as those pixels in `A` that match the values in vector `v`. `BW` is a binary image with 1's where the values of `A` match the values of `v`.

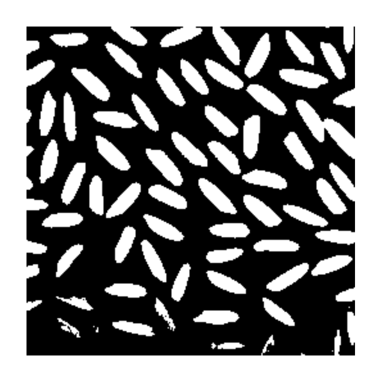
Class Support

The input image, `A`, must be numeric. The output image, `BW`, is of class `logical`.

Example

```
I = imread('rice.tif');
BW = roicolor(I,128,255);
imshow(I);
figure, imshow(BW)
```





See Also

`roifilt2`, `roipoly`

roifill

Purpose

Smoothly interpolate within an arbitrary image region

Syntax

```
J = roifill(I,c,r)
J = roifill(I)

J = roifill(I,BW)
[J,BW] = roifill(...)

J = roifill(x,y,I,xi,yi)
[x,y,J,BW,xi,yi] = roifill(...)
```

Description

`roifill` fills in a specified polygon in an intensity image. It smoothly interpolates inward from the pixel values on the boundary of the polygon by solving Laplace's equation. `roifill` can be used, for example, to "erase" small objects in an image.

`J = roifill(I,c,r)` fills in the polygon specified by `c` and `r`, which are equal-length vectors containing the row-column coordinates of the pixels on vertices of the polygon. The `k-th` vertex is the pixel `(r(k),c(k))`.

`J = roifill(I)` displays the image `I` on the screen and lets you specify the polygon using the mouse. If you omit `I`, `roifill` operates on the image in the current axes. Use normal button clicks to add vertices to the polygon. Pressing **Backspace** or **Delete** removes the previously selected vertex. A shift-click, right-click, or double-click adds a final vertex to the selection and then starts the fill; pressing **Return** finishes the selection without adding a vertex.

`J = roifill(I,BW)` uses `BW` (a binary image the same size as `I`) as a mask. `roifill` fills in the regions in `I` corresponding to the nonzero pixels in `BW`. If there are multiple regions, `roifill` performs the interpolation on each region independently.

`[J,BW] = roifill(...)` returns the binary mask used to determine which pixels in `I` get filled. `BW` is a binary image the same size as `I` with 1's for pixels corresponding to the interpolated region of `I` and 0's elsewhere.

`J = roifill(x,y,I,xi,yi)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system. `xi` and `yi` are equal-length vectors that specify polygon vertices as locations in this coordinate system.

[x,y,J,BW,xi,yi] = roifill(...) returns the XData and YData in x and y; the output image in J; the mask image in BW; and the polygon coordinates in xi and yi. xi and yi are empty if the roifill(I,BW) form is used.

If roifill is called with no output arguments, the resulting image is displayed in a new figure.

Class Support

The input image I can of class uint8, uint16, or double. The input binary mask, BW, can be any numeric class or logical. The output binary mask, BW, is always logical.The output image J is of the same class as I. All other inputs and outputs are of class double.

Example

```
I = imread('eight.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
J = roifill(I,c,r);
imshow(I)
figure, imshow(J)
```





See Also

roifilt2, roipoly

roifilt2

Purpose

Filter a region of interest

Syntax

```
J = roifilt2(h,I,BW)
J = roifilt2(I,BW,fun)
J = roifilt2(I,BW,fun,P1,P2,...)
```

Description

`J = roifilt2(h,I,BW)` filters the data in `I` with the two-dimensional linear filter `h`. `BW` is a binary image the same size as `I` that is used as a mask for filtering. `roifilt2` returns an image that consists of filtered values for pixels in locations where `BW` contains 1's, and unfiltered values for pixels in locations where `BW` contains 0's. For this syntax, `roifilt2` calls `filter2` to implement the filter.

`J = roifilt2(I,BW,fun)` processes the data in `I` using the function `fun`. The result `J` contains computed values for pixels in locations where `BW` contains 1's, and the actual values in `I` for pixels in locations where `BW` contains 0's.

`fun` can be a `function_handle`, created using `@`, or an inline object. `fun` should take a matrix as a single argument and return a matrix of the same size.

```
y = fun(x)
```

`J = roifilt2(I,BW,fun,P1,P2,...)` passes the additional parameters `P1,P2,...,` to `fun`.

Class Support

For the syntax that includes a filter, `h`, the input image, `I`, can be of class `uint8`, `uint16`, or `double`, and the output array, `J`, has the same class as the input image. For the syntax that includes a function, `I` can be of any class supported by `fun`, and the class of `J` depends on the class of the output from `fun`.

Example

This example continues the `roipoly` example.

```
I = imread('eight.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
BW = roipoly(I,c,r);
h = fspecial('unsharp');
J = roifilt2(h,I,BW);
imshow(J), figure, imshow(J)
```



See Also filter2, roipoly

Purpose

Select a polygonal region of interest

Syntax

```
BW = roipoly(I,c,r)
BW = roipoly(I)

BW = roipoly(x,y,I,xi,yi)
[BW,xi,yi] = roipoly(...)
[x,y,BW,xi,yi] = roipoly(...)
```

Description

Use `roipoly` to select a polygonal region of interest within an image. `roipoly` returns a binary image that you can use as a mask for masked filtering.

`BW = roipoly(I,c,r)` returns the region of interest selected by the polygon described by vectors `c` and `r`. `BW` is a binary image the same size as `I` with 0's outside the region of interest and 1's inside.

`BW = roipoly(I)` displays the image `I` on the screen and lets you specify the polygon using the mouse. If you omit `I`, `roipoly` operates on the image in the current axes. Use normal button clicks to add vertices to the polygon. Pressing **Backspace** or **Delete** removes the previously selected vertex. A shift-click, right-click, or double-click adds a final vertex to the selection and then starts the fill; pressing **Return** finishes the selection without adding a vertex.

`BW = roipoly(x,y,I,xi,yi)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system. `xi` and `yi` are equal-length vectors that specify polygon vertices as locations in this coordinate system.

`[BW,xi,yi] = roipoly(...)` returns the polygon coordinates in `xi` and `yi`. Note that `roipoly` always produces a closed polygon. If the points specified describe a closed polygon (i.e., if the last pair of coordinates is identical to the first pair), the length of `xi` and `yi` is equal to the number of points specified. If the points specified do not describe a closed polygon, `roipoly` adds a final point having the same coordinates as the first point. (In this case the length of `xi` and `yi` is one greater than the number of points specified.)

`[x,y,BW,xi,yi] = roipoly(...)` returns the `XData` and `YData` in `x` and `y`; the mask image in `BW`; and the polygon coordinates in `xi` and `yi`.

If `roipoly` is called with no output arguments, the resulting image is displayed in a new figure.

Class Support

The input image I can be of class uint8, uint16, or double. The output image BW is of class logical. All other inputs and outputs are of class double.

Remarks

For any of the roipoly syntaxes, you can replace the input image I with two arguments, m and n, that specify the row and column dimensions of an arbitrary image. For example, these commands create a 100-by-200 binary mask.

```
c = [112 112 79 79];
r = [37 66 66 37];
BW = roipoly(100,200,c,r);
```

If you specify m and n with an interactive form of roipoly, an m-by-n black image is displayed, and you use the mouse to specify a polygon within this image.

Example

```
I = imread('eight.tif');
c = [222 272 300 270 221 194];
r = [21 21 75 121 121 75];
BW = roipoly(I,c,r);
imshow(I)
figure, imshow(BW)
```





See Also

roifilt2, roicolor, roifill

std2

Purpose Compute the standard deviation of the elements of a matrix

Syntax

```
b = std2(A)
```

Description `b = std2(A)` computes the standard deviation of the values in `A`.

Class Support `A` can be numeric or logical. `B` is a scalar of class `double`.

Algorithm `std2` computes the standard deviation of the array `A` using `std(A(:))`.

See Also `corr2`, `mean2`

`std`, `mean` in the MATLAB Function Reference

Purpose

Create morphological structuring element

Syntax

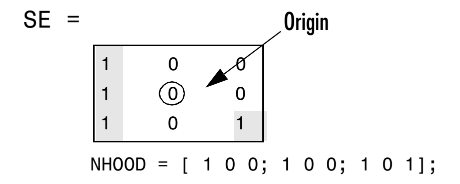
```
SE = strel(shape,parameters)
```

Description

`SE = strel(shape,parameters)` creates a structuring element, `SE`, of the type specified by *shape*. This table lists all the supported shapes. Depending on *shape*, `strel` may take additional parameters. See the syntax descriptions that follow for details about creating each type of structuring element.

Flat Structuring Elements	
`'arbitrary'`	`'pair'`
`'diamond'`	`'periodicline'`
`'disk'`	`'rectangle'`
`'line'`	`'square'`
`'octagon'`	
Nonflat Structuring Elements	
`'arbitrary'`	`'ball'`

`SE = strel('arbitrary',NHOOD)` creates a flat structuring element, where `NHOOD` specifies the neighborhood. `NHOOD` is a matrix containing 1's and 0's; the location of the 1's defines the neighborhood for the morphological operation. The center (or *origin*) of `NHOOD` is it's center element, given by `floor((size(NHOOD)+1)/2)`. You can omit the `'arbitrary'` string and just use `strel(NHOOD)`.



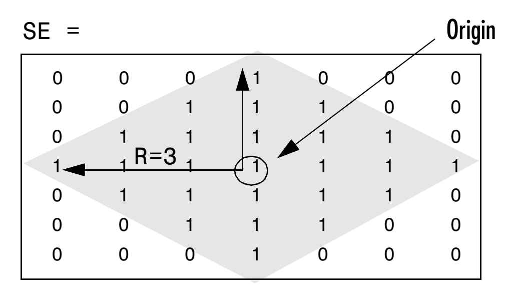
`SE = strel('arbitrary',NHOOD,HEIGHT)` creates a nonflat structuring element, where `NHOOD` specifies the neighborhood. `HEIGHT` is a matrix the same size as `NHOOD` containing the height values associated with each nonzero

element of NHOOD. The HEIGHT matrix must be real and finite valued. You can omit the 'arbitrary' string and just use strel(NHOOD,HEIGHT).

SE = strel('ball',R,H,N) creates a nonflat, "ball-shaped" (actually an ellipsoid) structuring element whose radius in the X-Y plane is R and whose height is H. Note that R must be a nonnegative integer, H must be a real scalar, and N must be an even nonnegative integer. When N is greater than 0, the ball-shaped structuring element is approximated by a sequence of N nonflat, line-shaped structuring elements. When N equals 0, no approximation is used, and the structuring element members comprise all pixels whose centers are no greater than R away from the origin. The corresponding height values are determined from the formula of the ellipsoid specified by R and H. If N is not specified, the default value is 8.

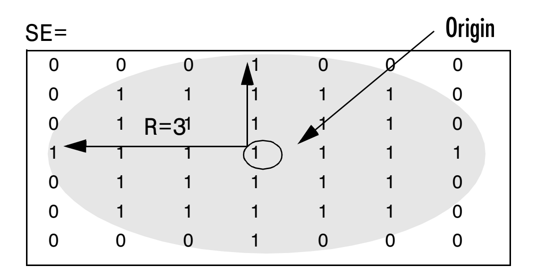
Note Morphological operations run much faster when the structuring element uses approximations (N>0) than when it does not (N=0).

SE = strel('diamond',R) creates a flat, diamond-shaped structuring element, where R specifies the distance from the structuring element origin to the points of the diamond. R must be a nonnegative integer scalar.

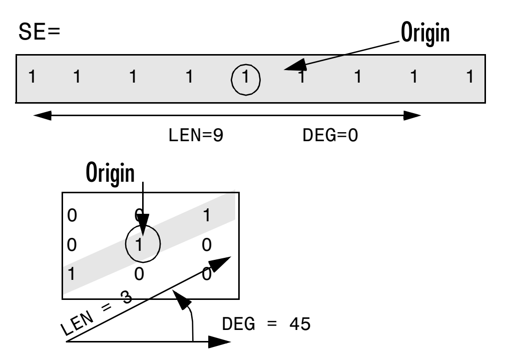


SE = strel('disk',R,N) creates a flat, disk-shaped structuring element, where R specifies the radius. R must be a nonnegative integer. N must be 0, 4, 6, or 8. When N is greater than 0, the disk-shaped structuring element is approximated by a sequence of N periodic-line structuring elements. When N equals 0, no approximation is used, and the structuring element members comprise all pixels whose centers are no greater than R away from the origin. If N is not specified, the default value is 4.

Note Morphological operations run much faster when the structuring element uses approximations (`N>0`) than when it does not (`N=0`). However, structuring elements that do not use approximations (`N=0`) are not suitable for computing granulometries. Sometimes it is necessary for `strel` to use two extra line structuring elements in the approximation, in which case the number of decomposed structuring elements used is `N+2`.

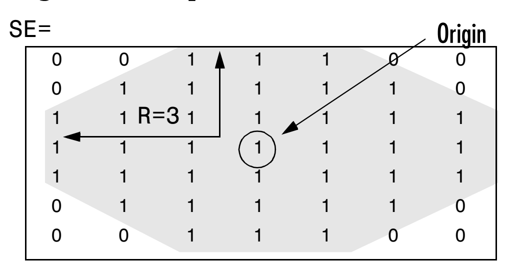


`SE = strel('line',LEN,DEG)` creates a flat, linear structuring element, where `LEN` specifies the length, and `DEG` specifies the angle (in degrees) of the line, as measured in a counterclockwise direction from the horizontal axis. `LEN` is approximately the distance between the centers of the structuring element members at opposite ends of the line.

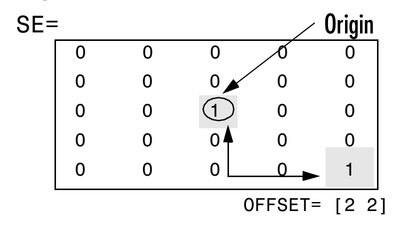


`SE = strel('octagon',R)` creates a flat, octagonal structuring element, where `R` specifies the distance from the structuring element origin to the sides of the

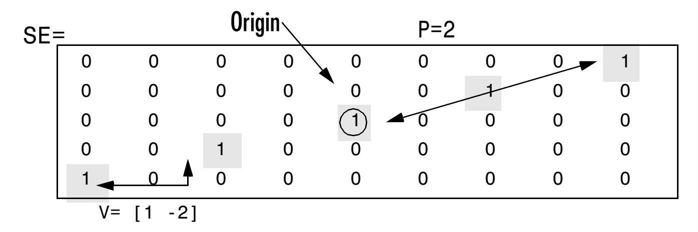
octagon, as measured along the horizontal and vertical axes. R must be a nonnegative multiple of 3.



SE = strel('pair',OFFSET) creates a flat structuring element containing two members. One member is located at the origin. The second member's location is specified by the vector OFFSET. OFFSET must be a two-element vector of integers.

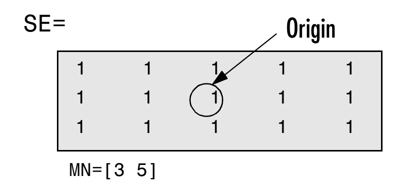


SE = strel('periodicline',P,V) creates a flat structuring element containing 2*P+1 members. V is a two-element vector containing integer-valued row and column offsets. One structuring element member is located at the origin. The other members are located at 1*V, -1*V, 2*V, -2*V, ..., P*V, -P*V.

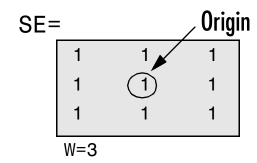


SE = strel('rectangle',MN) creates a flat, rectangle-shaped structuring element, where MN specifies the size. MN must be a two-element vector of nonnegative integers. The first element of MN is the number rows in the

structuring element neighborhood; the second element is the number of columns.



`SE = strel('square',W)` creates a square structuring element whose width is `W` pixels. `W` must be a nonnegative integer scalar.



Notes

For all shapes except `'arbitrary'`, structuring elements are constructed using a family of techniques known collectively as *structuring element decomposition*. The principle is that dilation by some large structuring elements can be computed faster by dilation with a sequence of smaller structuring elements. For example, dilation by an 11-by-11 square structuring element can be accomplished by dilating first with a 1-by-11 structuring element and then with an 11-by-1 structuring element. This results in a theoretical performance improvement of a factor of 5.5, although in practice the actual performance improvement is somewhat less. Structuring element decompositions used for the `'disk'` and `'ball'` shapes are approximations; all other decompositions are exact.

Methods

This table lists the methods supported by the `STREL` object.

`getheight`	Get height of structuring element
`getneighbors`	Get structuring element neighbor locations and heights
`getnhood`	Get structuring element neighborhood

getsequence	Extract sequence of decomposed structuring elements
isflat	Return true for flat structuring element
reflect	Reflect structuring element
translate	Translate structuring element

Example

```
se1 = strel('square',11)      % 11-by-11 square
se2 = strel('line',10,45)    % line, length 10, angle 45 degrees
se3 = strel('disk',15)        % disk, radius 15
se4 = strel('ball',15,5)      % ball, radius 15, height 5
```

Algorithm

The method used to decompose diamond-shaped structuring elements is known as “logarithmic decomposition” [1].

The method used to decompose disk structuring elements is based on the technique called “radial decomposition using periodic lines” [2], [3]. For details, see the `MakeDiskStrel` subfunction in `toolbox/images/images/@strel/strel.m`.

The method used to decompose ball structuring elements is the technique called “radial decomposition of sphere” [2].

See Also

`imdilate`, `imerode`

References

[1] Rein van den Boomgard and Richard van Balen, “Methods for Fast Morphological Image Transforms Using Bitmapped Images,” *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, vol. 54, no. 3, May 1992, pp. 252-254.

[2] Rolf Adams, “Radial Decomposition of Discs and Spheres,” *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, vol. 55, no. 5, September 1993, pp. 325-332.

[3] Ronald Jones and Pierre Soille, “Periodic lines: Definition, cascades, and application to granulometrie,” *Pattern Recognition Letters*, vol. 17, 1996, 1057-1063.

Purpose

Find limits to contrast stretch an image

Syntax

```
LOW_HIGH = stretchlim(I,TOL)
LOW_HIGH = stretchlim(RGB,TOL)
```

Description

`LOW_HIGH = stretchlim(I,TOL)` returns a pair of intensities that can be used by `imadjust` to increase the contrast of an image.

`TOL = [LOW_FRACT HIGH_FRACT]` specifies the fraction of the image to saturate at low and high intensities.

If `TOL` is a scalar, `TOL = LOW_FRACT`, and `HIGH_FRACT = 1 - LOW_FRACT`, which saturates equal fractions at low and high intensities.

If you omit the argument, `TOL` defaults to `[0.01 0.99]`, saturating 2%.

If `TOL = 0`, `LOW_HIGH = [min(I(:)) max(I(:))]`.

`LOW_HIGH = stretchlim(RGB,TOL)` returns a 2-by-3 matrix of intensity pairs to saturate each plane of the `RGB` image. `TOL` specifies the same fractions of saturation for each plane.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`. The output intensities returned, `LOW_HIGH`, are of class `double` and have values between 0 and 1.

Example

```
I = imread('pout.tif');
J = imadjust(I,stretchlim(I),[]);
imshow(I), figure, imshow(J)
```





See Also

`brighten`, `histeq`, `imadjust`

subimage

Purpose Display multiple images in the same figure

Syntax

```
subimage(X,map)
subimage(I)
subimage(BW)
subimage(RGB)
subimage(x,y,...)
h = subimage(...)
```

Description You can use `subimage` in conjunction with `subplot` to create figures with multiple images, even if the images have different colormaps. `subimage` works by converting images to truecolor for display purposes, thus avoiding colormap conflicts.

`subimage(X,map)` displays the indexed image `X` with colormap `map` in the current axes.

`subimage(I)` displays the intensity image `I` in the current axes.

`subimage(BW)` displays the binary image `BW` in the current axes.

`subimage(RGB)` displays the truecolor image `RGB` in the current axes.

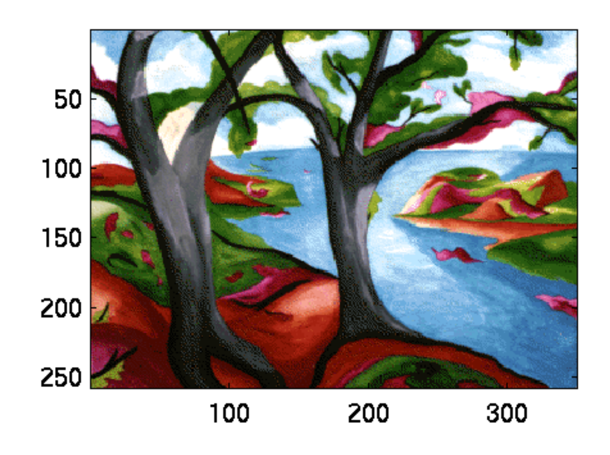
`subimage(x,y...)` displays an image using a nondefault spatial coordinate system.

`h = subimage(...)` returns a handle to an image object.

Class Support The input image can be of class `logical`, `uint8`, `uint16`, or `double`.

Example

```
load trees
[X2,map2] = imread('forest.tif');
subplot(1,2,1), subimage(X,map)
subplot(1,2,2), subimage(X2,map2)
```





See Also `imshow`

`subplot` in the MATLAB Function Reference

Purpose

Spatial transformation of a multidimensional array

Syntax

```
B = tformarray(A,T,R,TDIMS_A,TDIMS_B,TSIZE_B,TMAP_B,F)
```

Description

`B = tformarray(A,T,R,TDIMS_A,TDIMS_B,TSIZE_B,TMAP_B,F)` applies a spatial transformation to array `A` to produce array `B`. The `tformarray` function is like `imtransform`, but is intended for problems involving higher-dimensioned arrays or mixed input/output dimensionality, or requiring greater user control or customization. (Anything that can be accomplished with `imtransform` can be accomplished with a combination of `maketform`, `makeresampler`, `findbounds`, and `tformarray`; but for many tasks involving 2-D images, `imtransform` is simpler.)

This table provides a brief description of all the input arguments. See the following section for more detail about each argument. (Click on an argument in the table to move to the appropriate section.)

Argument	Description
`A`	Input array or image
`T`	Spatial transformation structure, called a `TFORM`, typically created with `maketform`
`R`	Resampler structure, typically created with `makeresampler`
`TDIMS_A`	Row vector listing the input transform dimensions
`TDIMS_B`	Row vector listing the output transform dimensions
`TMAP_B`	Output array size in the transform dimensions
`TSIZE_B`	Array of point locations in output space; can be used as an alternative way to specify a spatial transformation
`F`	Array of fill values

`A` can be any nonsparse numeric array, and can be real or complex.

T is a TFORM structure that defines a particular spatial transformation. For each location in the output transform subscript space (as defined by TDIMS_B and TSIZE_B), tformarray uses T and the function tforminv to compute the corresponding location in the input transform subscript space (as defined by TDIMS_A and size(A)).

If T is empty, tformarray operates as a direct resampling function, applying the resampler defined in R to compute values at each transform space location defined in TMAP_B (if TMAP_B is non-empty), or at each location in the output transform subscript grid.

R is a structure that defines how to interpolate values of the input array at specified locations. R is usually created with makeresampler, which allows fine control over how to interpolate along each dimension, as well as what input array values to use when interpolating close the edge of the array.

TDIMS_A and TDIMS_B indicate which dimensions of the input and output arrays are involved in the spatial transformation. Each element must be unique, and must be a positive integer. The entries need not be listed in increasing order, but the order matters. It specifies the precise correspondence between dimensions of arrays A and B and the input and output spaces of the transformer, T. length(TDIMS_A) must equal T.ndims_in, and LENGTH(TDIMS_B) must equal T.ndims_out.

For example, if T is a 2-D transformation, TDIMS_A = [2 1], and TDIMS_B = [1 2], then the column dimension and row dimension of A correspond to the first and second transformation input-space dimensions, respectively. The row and column dimensions of B correspond to the first and second output-space dimensions, respectively.

TSIZE_B specifies the size of the array B along the output-space transform dimensions. Note that the size of B along nontransform dimensions is taken directly from the size of A along those dimensions. If, for example, T is a 2-D transformation, size(A) = [480 640 3 10], TDIMS_B is [2 1], and TSIZE_B is [300 200], then size(B) is [200 300 3].

TMAP_B is an optional array that provides an alternative way of specifying the correspondence between the position of elements of B and the location in output transform space. TMAP_B can be used, for example, to compute the result of an image warp at a set of arbitrary locations in output space. If TMAP_B is not empty, then the size of TMAP_B takes the form:

```
[D1 D2 D3 ... DN L]
```

where N equals length(TDIMS_B). The vector [D1 D2 ... DN] is used in place of TSIZE_B. If TMAP_B is not empty, then TSIZE_B should be [].

The value of L depends on whether or not T is empty. If T is not empty, then L is T.ndims_out, and each L-dimension point in TMAP_B is transformed to an input-space location using T. If T is empty, then L is length(TDIMS_A), and each L-dimensional point in TMAP_B is used directly as a location in input space.

F is a double-precision array containing fill values. The fill values in F may be used in three situations:

- When a separable resampler is created with makeresampler and its padmethod is set to either 'fill' or 'bound'.
- When a custom resampler is used that supports the 'fill' or 'bound' pad methods (with behavior that is specific to the customization).
- When the map from the transform dimensions of B to the transform dimensions of A is deliberately undefined for some points. Such points are encoded in the input transform space by NaNs in either TMAP_B or in the output of TFORMINV.

In the first two cases, fill values are used to compute values for output locations that map outside or near the edges of the input array. Fill values are copied into B when output locations map well outside the input array. See makeresampler for more information about 'fill' and 'bound'.

F can be a scalar (including NaN), in which case its value is replicated across all the nontransform dimensions. F can also be a nonscalar, whose size depends on size(A) in the nontransform dimensions. Specifically, if K is the J-th nontransform dimension of A, then size(F,J) must be either size(A,K) or 1. As a convenience to the user, tformarray replicates F across any dimensions with unit size such that after the replication size(F,J) equals size(A,K).

For example, suppose A represents 10 RGB images and has size 200-by-200-by-3-by-10, T is a 2-D transformation, and TDIMS_A and TDIMS_B are both [1 2]. In other words, tformarray will apply the same 2-D transform to each color plane of each of the 10 RGB images. In this situation you have several options for F:

- F can be a scalar, in which case the same fill value is used for each color plane of all 10 images.
- F can be a 3-by-1 vector, [R G B]'. Then R, G, and B will be used as the fill values for the corresponding color planes of each of the 10 images. This can be interpreted as specifying an RGB "fill color," with the same color used for all 10 images.
- F can be a 1-by-10 vector. This can be interpreted as specifying a different fill value for each of 10 images, with that fill value being used for all three color planes.
- F can be a 3-by-10 matrix, which can be interpreted as supplying a different RGB fill-color for each of the 10 images.

Class Support

A can be any nonsparse numeric array, and can be real or complex. It can also be of class `logical`.

Example

Create a 2-by-2 checkerboard image where each square is 20 pixels wide, then transform it with a projective transformation. Use a pad method of `'circular'` when creating a resampler, so that the output appears to be a perspective view of an infinite checkerboard. Swap the output dimensions. Specify a 100-by-100 output image. Leave `TMAP_B` empty, since `TSIZE_B` is specified. Leave the fill value empty, since it won't be needed.

```
I = checkerboard(20,1,1);
figure; imshow(I)
T = maketform('projective',[1 1; 41 1; 41 41;   1 41],...
                           [5 5; 40 5; 35 30; -10 30]);
R = makeresampler('cubic','circular');
J = tformarray(I,T,R,[1 2],[2 1],[100 100],[],[]);
figure; imshow(J)
```

See Also

`findbounds`, `imtransform`, `makeresampler`, `maketform`

tformfwd

Purpose

Apply forward spatial transformation

Syntax

```
X = tformfwd(U,T)
```

Description

X = tformfwd(U,T) applies the forward transformation defined in T to each row of U, if U is a matrix. T is a TFORM struct created with maketform, fliptform, or cp2tform. The argument U is an M-by-T.ndims_in matrix. Each row of U represents a point in T's input space. The argument X is an M-by-T.ndims_out matrix. Each row of X represents a point in T's output space, and is the forward transformation of the corresponding row of U.

If U is an (N+1)-dimensional array (including an implicit trailing singleton dimension if T.ndims_in is 1), tformfwd applies the transformation to each point U(P1,P2,...,PN,:). The value size(U,N+1) must equal T.ndims_in. The returned array, X, is an (N+1)-dimensional array containing the results of each transformation in X(P1,P2,...PN,:). The value size(X,I) equals size(U,I) for I=1,...,N and size(X,N+1) equals T.ndims_out.

Example

Create an affine transformation that maps the triangle with vertices (0,0), (6,3), (-2,5) to the triangle with vertices (-1,-1), (0,-10), (-2,5).

```
u = [0 0; 6 3; -2 5];
x = [-1 -1; 0 -10; 4 4];
tform = maketform('affine',u,x);
```

Validate the mapping by applying tformfwd to u.

```
tformfwd(u,tform)  % Result should equal x
```

See Also

cp2tform, fliptform, maketform, tforminv

tforminv

Purpose Apply inverse spatial transformation

Syntax

```
U = tforminv(X,T)
```

Description

`U = tforminv(X,T)` applies the inverse transformation defined in `T` to each row of `X`, if `X` is a matrix. `T` is a `TFORM` struct created with `maketform`, `fliptform`, or `cp2tform`. The argument `X` is an `M`-by-`T.ndims_out` matrix. Each row of `X` represents a point in `T`'s output space. `U` is an `M`-by-`T.ndims_out` matrix. Each row of `U` represents a point in `T`'s input space, and is the inverse transformation of the corresponding row of `X`.

If `X` is an (N+1)-dimensional array (including an implicit trailing singleton dimension if `T.ndims_out` is 1), then `tforminv` applies the transformation to each point `X(P1,P2,...,PN,:)`. `size(X,N+1)` must equal `T.ndims_in`. `U` is an (N+1)-dimensional array containing the results of each transformation in `U(P1,P2,...PN,:)`. `size(U,I)` equals `size(X,I)` for `I = 1,...,N` and `size(U,N+1)` equals `T.ndims_in`.

Example

Create an affine transformation that maps the triangle with vertices (0,0), (6,3), (-2,5) to the triangle with vertices (-1,-1), (0,-10), (-2,5).

```
u = [0 0; 6 3; -2 5];
x = [-1 -1; 0 -10; 4 4];
tform = maketform('affine',u,x);
```

Validate the mapping by applying `tforminv` to x.

```
tforminv(x,tform)  % Result should equal u
```

See Also

`cp2tform`, `tforminv`, `maketform`, `fliptform`

translate

Purpose

Translate structuring element

Syntax

```
SE2 = translate(SE,V)
```

Description

`SE2 = reflect(SE,V)` translates a structuring element `SE` in N-D space. `V` is an N-element vector containing the offsets of the desired translation in each dimension.

Class Support

`SE` and `SE2` are `STREL` objects; `V` is a vector of double precision values.

Example

Dilating with a translated version of `strel(1)` is a way to translate the input image in space. This example translates the `cameraman.tif` image down and to the right by 25 pixels.

```
I = imread('cameraman.tif');
se = translate(strel(1), [25 25]);
J = imdilate(I,se);
imshow(I), title('Original')
figure, imshow(J), title('Translated');
```



See Also

`strel`, `reflect`

truesize

Purpose

Adjust display size of an image

Syntax

```
truesize(fig,[mrows mcols])
truesize(fig)
```

Description

`truesize(fig,[mrows ncols])` adjusts the display size of an image. `fig` is a figure containing a single image or a single image with a colorbar. `[mrows ncols]` is a 1-by-2 vector that specifies the requested screen area in pixels that the image should occupy.

`truesize(fig)` uses the image height and width for `[mrows ncols]`. This results in the display having one screen pixel for each image pixel.

If you omit the figure argument, `truesize` works on the current figure.

Remarks

If the `'TruesizeWarning'` toolbox preference is `'on'`, `truesize` displays a warning if the image is too large to fit on the screen. (The entire image is still displayed, but at less than true size.) If `'TruesizeWarning'` is `'off'`, `truesize` does not display the warning. Note that this preference applies even when you call `truesize` indirectly, such as through `imshow`.

See Also

`imshow`, `iptsetpref`, `iptgetpref`

Purpose

Convert data to unsigned 8-bit integers

`uint8` is a MATLAB built-in function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

uint16

Purpose Convert data to unsigned 16-bit integers

`uint16` is a MATLAB built-in function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Purpose Display an image as a texture-mapped surface

Syntax

```
warp(X,map)
warp(I,n)
warp(BW)
warp(RGB)
warp(z,...)
warp(x,y,z,...)
h = warp(...)
```

Description `warp(X,map)` displays the indexed image `X` with colormap `map` as a texture map on a simple rectangular surface.

`warp(I,n)` displays the intensity image `I` with gray scale colormap of length `n` as a texture map on a simple rectangular surface.

`warp(BW)` displays the binary image `BW` as a texture map on a simple rectangular surface.

`warp(RGB)` displays the RGB image in the array `RGB` as a texture map on a simple rectangular surface.

`warp(z,...)` displays the image on the surface `z`.

`warp(x,y,z...)` displays the image on the surface `(x,y,z)`.

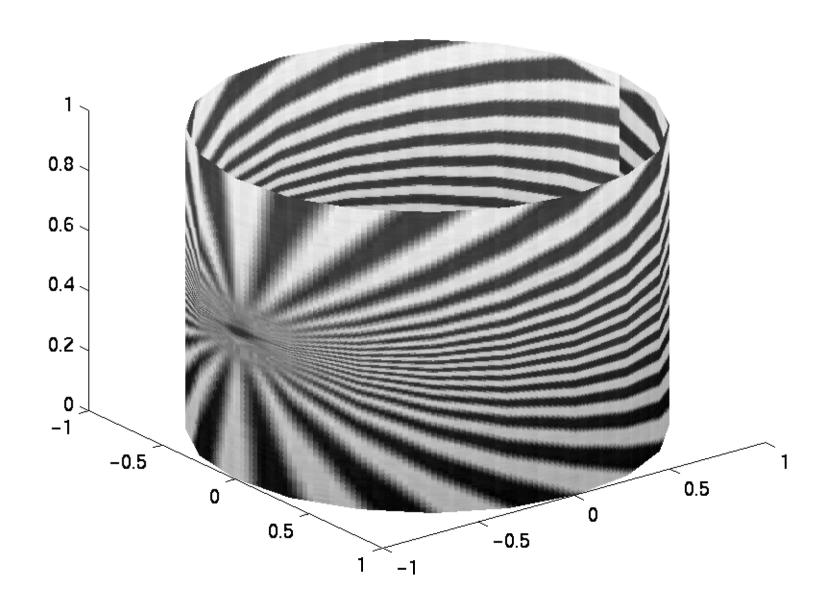
`h = warp(...)` returns a handle to a texture mapped surface.

Class Support The input image can be of class `logical`, `uint8`, `uint16`, or `double`.

Remarks Texture-mapped surfaces generally render more slowly than images.

Example This example texture maps an image of a test pattern onto a cylinder.

```
[x,y,z] = cylinder;
I = imread('testpat1.tif');
warp(x,y,z,I);
```



See Also

`imshow`

`image`, `imagesc`, `surf` in the MATLAB Function Reference

Purpose

Find image watershed regions

Syntax

```
L = watershed(A)
L = watershed(A,CONN)
```

Description

`L = watershed(A)` computes a label matrix identifying the watershed regions of the input matrix `A`, which can have any dimension. The elements of `L` are integer values greater than or equal to 0. The elements labeled 0 do not belong to a unique watershed region. These are called *watershed pixels*. The elements labeled 1 belong to the first watershed region, the elements labeled 2 belong to the second watershed region, and so on.

By default, `watershed` uses 8-connected neighborhoods for 2-D inputs and 26-connected neighborhoods for 3-D inputs. For higher dimensions, `watershed` uses the connectivity given by `conndef(ndims(A),'maximal')`.

`L = watershed(A,CONN)` specifies the connectivity to be used in the watershed computation. `CONN` may have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
`4`	4-connected neighborhood
`8`	8-connected neighborhood
Three-dimensional connectivities	
`6`	6-connected neighborhood
`18`	18-connected neighborhood
`26`	26-connected neighborhood

Connectivity may be defined in a more general way for any dimension by using for `CONN` a 3-by-3-by- ...-by-3 matrix of `0`'s and `1`'s. The `1`-valued elements define neighborhood locations relative to the center element of `CONN`. Note that `CONN` must be symmetric about its center element.

Class Support

A can be a numeric or logical array of any dimension, and it must be nonsparse. The output array L is of class double.

Example

2-D Example:

1 Make a binary image containing two overlapping circular objects.

```
center1 = -10;
center2 = -center1;
dist = sqrt(2*(2*center1)^2);
radius = dist/2 * 1.4;
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];
[x,y] = meshgrid(lims(1):lims(2));
bw1 = sqrt((x-center1).^2 + (y-center1).^2) <= radius;
bw2 = sqrt((x-center2).^2 + (y-center2).^2) <= radius;
bw = bw1 | bw2;
figure, imshow(bw,'n'), title('bw')
```

2 Compute the distance transform of the complement of the binary image.

```
D = bwdist(~bw);
figure, imshow(D,[],'n'), title('Distance transform of ~bw')
```

3 Complement the distance transform, and force pixels that don't belong to the objects to be at -Inf.

```
D = -D;
D(~bw) = -Inf;
```

4 Compute the watershed transform and display it as an indexed image.

```
L = watershed(D);
rgb = label2rgb(L,'jet',[.5 .5 .5]);
figure, imshow(rgb,'n'), title('Watershed transfrom of D');
```

3-D Example:

1 Make a 3-D binary image containing two overlapping spheres.

```
center1 = -10;
center2 = -center1;
dist = sqrt(3*(2*center1)^2);
radius = dist/2 * 1.4;
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];
[x,y,z] = meshgrid(lims(1):lims(2));
```

```
bw1 = sqrt((x-center1).^2 + (y-center1).^2 + ...
     (z-center1).^2) <= radius;
bw2 = sqrt((x-center2).^2 + (y-center2).^2 + ...
     (z-center2).^2) <= radius;
bw = bw1 | bw2;
figure, isosurface(x,y,z,bw,0.5), axis equal, title('BW')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
```

2 Compute the distance transform.

```
D = bwdist(~bw);
figure, isosurface(x,y,z,D,radius/2), axis equal
title('Isosurface of distance transform')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
```

3 Complement the distance transform, force nonobject pixels to be `-Inf`, and then compute the watershed transform.

```
D = -D;
D(~bw) = -Inf;
L = watershed(D);
figure, isosurface(x,y,z,L==2,0.5), axis equal
title('Segmented object')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
figure, isosurface(x,y,z,L==3,0.5), axis equal
title('Segmented object')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud
```

Algorithm

`watershed` uses a variation of the Vincent and Soille algorithm [1]. For details of the variation, see `toolbox/images/images/private/watershed_vs.h`.

See Also

`bwlabel`, `bwlabeln`, `bwdist`, `regionprops`

Reference

[1] Luc Vincent and Pierre Soille, “Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations,” *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. 13, no. 6, June 1991, pp. 583-598.

Purpose

Perform two-dimensional adaptive noise-removal filtering

Syntax

```
J = wiener2(I,[m n],noise)
[J,noise] = wiener2(I,[m n])
```

Description

`wiener2` lowpass filters an intensity image that has been degraded by constant power additive noise. `wiener2` uses a pixel-wise adaptive Wiener method based on statistics estimated from a local neighborhood of each pixel.

`J = wiener2(I,[m n],noise)` filters the image `I` using pixel-wise adaptive Wiener filtering, using neighborhoods of size `m`-by-`n` to estimate the local image mean and standard deviation. If you omit the `[m n]` argument, `m` and `n` default to 3. The additive noise (Gaussian white noise) power is assumed to be `noise`.

`[J,noise] = wiener2(I,[m n])` also estimates the additive noise power before doing the filtering. `wiener2` returns this estimate in `noise`.

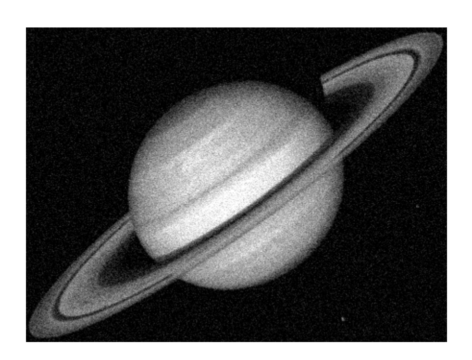
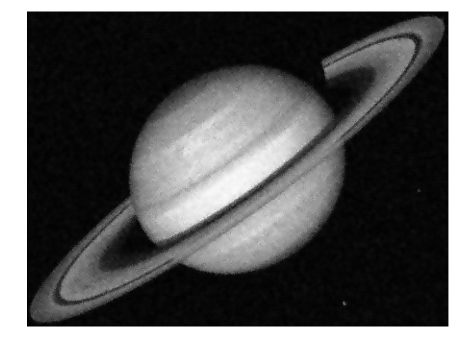
Class Support

The input image, `I`, is a two-dimensional image of class `uint8`, `uint16`, or `double`. The output image, `J`, is of the same size and class as `I`.

Example

Degrade and then restore an intensity image using adaptive Wiener filtering.

```
I = imread('saturn.tif');
J = imnoise(I,'gaussian',0,0.005);
K = wiener2(J,[5 5]);
imshow(J)
figure, imshow(K)
```

Algorithm

`wiener2` estimates the local mean and variance around each pixel

$$\mu = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a(n_1, n_2)$$

$$\sigma^2 = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a^2(n_1, n_2) - \mu^2$$

where η is the *N*-by-*M* local neighborhood of each pixel in the image `A`. `wiener2` then creates a pixel-wise Wiener filter using these estimates

$$b(n_1, n_2) = \mu + \frac{\sigma^2 - \nu^2}{\sigma^2}(a(n_1, n_2) - \mu)$$

where ν^2 is the noise variance. If the noise variance is not given, `wiener2` uses the average of all the local estimated variances.

See Also

`filter2`, `medfilt2`

Reference

[1] Lim, Jae S. *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1990. pp. 536-540.

Purpose Convert YCbCr values to RGB color space

Syntax

```
rgbmap = ycbcr2rgb(ycbcrmap)
RGB = ycbcr2rgb(YCBCR)
```

Description `rgbmap = ycbcr2rgb(ycbcrmap)` converts the YCbCr values in the colormap `ycbcrmap` to the RGB color space. If `ycbcrmap` is `m`-by-3 and contains the YCbCr luminance (*Y*) and chrominance (*Cb* and *Cr*) color values as columns, then `rgbmap` is returned as an `m`-by-3 matrix that contains the red, green, and blue values equivalent to those colors.

`RGB = ycbcr2rgb(YCBCR)` converts the YCbCr image `YCBCR` to the equivalent truecolor image `RGB`.

Class Support If the input is a YCbCr image, it can be of class `uint8`, `uint16`, or `double`; the output image is of the same class as the input image. If the input is a colormap, the input and output colormaps are both of class `double`.

See Also `ntsc2rgb`, `rgb2ntsc`, `rgb2ycbcr`

Purpose Zoom in and out on an image

`zoom` is a MATLAB function. To get help for this function, select **MATLAB Help** from the Help menu and view the online function reference page.

Index

Numerics

A

B

C

D

J

L

M

N

O

P

S

T